National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

NL-339 (r. 88/04) c

Canada

The University of Alberta

Functional Logic Programming

by

Kris Ng

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Masters of Science

Department of Computing Science

Edmonton, Alberta
Spring 1990

National Library of Canada
Canadian Theses Service
Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Service des thèses canadiennes

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

ISBN 0-315-60313-5

NL-339 (r. 88/04) c

Canada

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Kris Ng

TITLE OF THESIS: Functional Logic Programming

DEGREE: Masters of Science

YEAR THIS DEGREE GRANTED: 1990

Permission is hereby granted to THE UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed) .....

Permanent Address:
P.O.Box 369, Sub 11,
Edmonton, Alberta,
T6G 2E0, Canada.

Date: Nov 10, 1989

I do not know
what I may appear to the world,
but to myself
I seem to have been only like a boy
playing on the sea-shore,
and diverting myself in now and then
finding a smoother pebble
or a prettier shell than ordinary,
whilst the greatest ocean of truth
lay all undiscovered before me.

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **Functional Logic Programming** submitted by **Kris Ng** in partial fulfillment of the requirements for the degree of Masters of Science.

Jia-Huai You

Dr. Jia-Huai You (Supervisor)

R. Goebel

Dr. Randy Goebel

Ao Wai Hung

Dr. Joseph So

Li Yan Yuan

Dr. Li-Yan Yuan

Date: Nov 10, 1989

Dedicated to My Parents

Abstract

As a dialect of declarative languages, logic programming possesses the distinct property of separating declarative semantics from operational semantics. Despite its prominent features such as logical variables, bi-directional data flow and functional invertibility, the runtime behavior of non-deterministic logic programs is far too complicated and too difficult to control. Another major shortcoming of logic programming paradigm is the utilization of 'flat-natured' first-order constructor terms. With function evaluation, higher-order programming is possible for functional languages, whilst the expressive power of first-order logic languages in this count is severely restricted. Attention is drawn to integrate logic programming with a restricted class of first-order functional programming through an explicit treatment of equality. In this thesis, we propose a conditional term rewriting system which serves as the bridge to combine these two paradigms in a purely logical framework. The completeness and minimality characteristics of such system under a subset of classic equality, namely E_c-equality, are further investigated. A major contribution of this proposed system is the existence of an efficient implementation scheme which is readily portable to existing Prolog systems.

Acknowledgements

I would like to express my sincere gratitude towards my supervisor, Dr. Jia You, for his guidance and support throughout the course of this research.

I would also like to thank the members of my examining committee: Dr. Randy Goebel, Dr. Joseph So and Dr. Li-Yan Yuan for their careful reviewing and constructive comments. I also thank Ms. Kitty Leong for proof-reading my thesis, and Mr. Gordon Fong for many valuable discussions.

I am very grateful to my family members for their spiritual and financial support. Finally, I would like to express my deepest appreciation for all my friends in Edmonton for making my stay here a memorable one.

Table Of Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Objectives . . . 3
- 1.3 Theory of Logic Programming . . . 5
  - 1.3.1 Syntax . . . 5
  - 1.3.2 Semantics . . . 8
  - 1.3.3 Procedural Interpretation . . . 10

2 Functional Logic Languages **16**
- 2.1 Logic versus Functional Programming . . . 16
- 2.2 Operational Semantics . . . 21
  - 2.2.1 Preliminaries . . . 22
  - 2.2.2 Reduction . . . 25
  - 2.2.3 Narrowing . . . 27
- 2.3 Narrowing Strategies . . . 29

3 E_c-Equality Theory **34**
- 3.1 Equality Theory . . . 34
- 3.2 E_c-Equality . . . 37
- 3.3 Complete set of Minimal E_c-unifiers . . . 41
- 3.4 Outer-narrowing . . . 45

4 Conditional Rewriting System **55**
- 4.1 Syntax . . . 55
- 4.2 Completeness . . . 57
  - 4.2.1 E-equality vs E_c-equality . . . 61

4.3 Extended Unification 67

5 Functional Logic Programming **79**
5.1 Overview of WUP 80
5.1.1 Memory Management 81
5.1.2 Data Structure 82
5.1.3 Control Mechanism 85
5.2 Implementation of FLP 89

6 Conclusion **100**
6.1 Summary 100
6.2 Future research 103

Bibliography **106**

List of Figures

1.1 A trivial Prolog program 8
1.2 AND/OR proof tree of a logic program execution 15

3.1 Digraph of complete and minimal set of E_c-unifiers 44
3.2 Residue Map 49

4.1 Syntactic unification without occur check 68
4.2 Naive ABC algorithm 70
4.3 Extended unification scheme 72
4.4 Solver algorithm 73
4.5 Improvement over cross-unification 78

5.1 Declaration and molecular structure of PC_WORD 83
5.2 PC_WORD representation of a clause head 84
5.3 Memory layout of copy stack and runtime stack 86
5.4 Deterministic and non-deterministic stack frames 89
5.5 Unification table 90
5.6 PC_WORD representation of a modified clause head 91
5.7 Modified version of runtime stack frames 93
5.8 Simplified version of functor conflict handling 95
5.9 Pushing a pseudo stack node onto runtime stack 96
5.10 Modified proof procedure to handle rewritten terms 97
5.11 Runtime stack simulation of a program execution 99

Chapter 1

Introduction

This chapter outlines the motivation and objectives of the thesis, and some basic concepts of logic programming. The overview presented is far from complete and it only serves the purpose of allowing the thesis to be self-contained. Interested readers are referred to [Hog84, Kow74, Kow79, Llo84] for details of logic programming.

1.1 Motivation

Based on Von Neumann's model of computing, the center stage of conventional computer architecture is the sequential data flow between a central processing unit and its memory. Imperative programming languages tailored for such models suffer from what has been called "Von Neumann

Bottleneck". Since each program state is dependent on the state of its associated memory cells, exploitation of parallel execution is hindered by the sequential data transfer inherited from its underlying computer architecture. More critically, the control components used to alter program states must be embedded and mingled with their logic counterparts, resulting in a drastic increase of programming complexity. When compared with imperative languages, the most appealing feature of declarative programming languages is the distinction between *what* the output of a given program will be and *how* the output is obtained. Decoupling logic from control reduces programming complexity, promises higher expressive power and improves the possibility of parallelism. Two major schools of declarative languages, *Logic Programming* and *Functional Programming*, lay their foundations on sound mathematical logic to simplify their syntactic and semantic constructs. The most popular form of logic programming is based on first-order logic while functional programming traces its origin to lambda calculus. Even though these two paradigms exhibit different programming properties, each of them provides many exclusive but complementary features which makes it worthwhile to combine them under one roof. In particular, lambda calculus can be dealt with as a first order theory with equality. Attention is drawn to formalize both in a purely logical framework, which combines logic programming with a restricted class of first-order functional programming through an explicit treatment of equality.

The technique for solving systems of equations has been commonly used in equational logic programming. In the context of equation solving, two non-ground terms are unifiable if there exists a variable substitution such that the substituted terms are equaled in a well-defined framework. This seemingly simple definition can be viewed as a notion of E-unification [Plo72]with two open-ended questions:

- How to establish the equivalence between different terms?
- What kind of framework is suitable for proving equality of two different terms?

1.2 Objectives

Equivalence between different terms is traditionally defined by an equational theory, which in turn, consists of a set of equations. The left hand side (L.H.S) of each equation is 'semantically the same as' its corresponding right hand side (R.H.S). Equation solving can now be characterized as unification modulo an equational theory, through a prescribed operational semantics. Recent research in amalgamating functional and logic programming has emphasized the existence of a complete and efficient operational semantics for certain restricted classes of equational or functional languages. Almost all these proposed languages rest upon the classical first-order equality axioms

of reflexivity, symmetry, transitivity and substitutivity. Inspired by the latter question, another area of research has been focused on non-trivial equality theory opulent enough to capture the intuition of first-order functional programming so that an efficient implementation exists.

The work attempted in this thesis is divided into two parts. The first part (Chapter 2, 3) is an overview of various aspects concerning combination of logic and functional programming. Chapter 2 presents current issues regarding the operational semantics for a resulting language. Chapter 3 is devoted to the discussion of a restricted equality theory, namely E_c-*equality* [You88], and an efficient reasoning mechanism for this particular equality theory. Chapter 4 and 5 constitute the latter part of the thesis, which proposes a conditional term rewriting system under E_c-equality. The operational semantics is a version of extended unification based on *Outer-Narrowing* [You88]. The primary objectives of the proposal are:

1. to investigate the completeness and minimality of the proposed conditional term rewriting system under E_c-equality.

2. to define an extended unification mechanism for such system.

3. to define an efficient implementation scheme which can be easily incorporated into existing Prolog interpreters/compilers.

1.3 Theory of Logic Programming

As a direct outgrowth of research in automatic theorem proving, in particular the development of *Resolution Principle* [Rob65], the fundamental idea of logic programming is to materialize the concept of using "predicate logic as a programming language with sound theoretical foundations" [Kow74]. Realization of logic programming language as a tool to axiomatize and infer over a given problem domain is justified by its special features such as separation of declarative and operational semantics, non-deterministic program execution and manipulation of partial data structures through logical variables. Its intelligibility is further reinforced by the procedural interpretation of Horn-clause subset of first-order predicate logic. Although we are concentrated on Horn clause logic in this thesis, the notion of logic programming is by no means restricted to this modest subset. Horn clause programming must be extended in various directions in order to fully compile with the broader view of logic programming: deduction as operational semantics for programs represented by formulae of a well-defined logical system.

1.3.1 Syntax

In the programming environment, the usual meaning of syntax refers to the way in which elements of a specific alphabet are put together to form an

admissible language construct. Similar to first-order theory, an alphabet $\mathcal{A}$ consists of four mutually disjointed classes of symbols: variables, constants, functors and predicates. The building blocks of logic programs are *terms* and *atoms*. A term is either a variable, a constant or inductively defined as an n-ary function $f(t_1, \ldots, t_n)$ where 'f' is a functor and $t_1, \ldots, t_n$ $(n \geq 1)$ are terms. An atom is an n-ary predicate $p(t_1, \ldots, t_n)$ where 'p' is a predicate symbol and $t_1, \ldots, t_n$ $(n \geq 1)$ are terms. The basic syntactic constructs of (generalized) first-order logic programs are clauses of the form

$$H_1, \ldots, H_m \leftarrow B_1, \ldots, B_n$$

where

$H_1, \ldots, H_m$ and $B_1, \ldots, B_n$ $(m \geq 0, n \geq 0)$ are atoms.

Each clause is implicitly quantified by an universal quantifier, so that the aforementioned clause is equivalent to

$$\forall \bar{x} \;\; H_1, \ldots, H_m \leftarrow B_1, \ldots, B_n$$

where

$\bar{x}$ is the set of free variables appearing in the clause.

The scope of a variable is bound to the clause containing it. Speaking of Horn clause programming, Horn clauses are restricted to those clauses

with at most one atom on the left hand side of the connective symbol "←" (i.e. with m $\leq$ 1). These clauses can be further classified into two categories, program clauses and goal clauses. A program clause takes the form of

$$H \leftarrow B_1, \ldots, B_n \qquad (n \geq 0)$$

The atom H is called the *conclusion* or *head* of the clause, whilst $B_1, \ldots, B_n$ are collectively referred as the *antecedent* or *body* of the clause. Program clauses are also known as *rules*, whereas special cases of such clauses with empty body (i.e. n = 0) are known as *assertions*. A set of program clauses, whose heads are atoms with the same predicate symbol and arity, define a *procedure* for that particular predicate symbol. A logic program is just a set of program clauses (or procedures) whose inference system is triggered by a given goal clause. A goal clause has the form of

$$\leftarrow B_1, \ldots, B_n \qquad (n \geq 0)$$

Each B_i (i = 1,...,n) is called a subgoal of the goal clause. An *empty* or *null* clause is a special goal clause with no body at all. Null clauses are traditionally interpreted as contradictions. Fig 1.1 is an example of a trivial Prolog program. The predicate grandson(X,Y) denotes that X is the grandson of Y, and son(X,Y) means that X is the son of Y.

```
grandson(X, Y) ← son(X, Z), son(Z, Y)
son(Joseph,Terry)
son(Terry,Louis)

  ? grandson(Joseph,Y)
```

Figure 1.1: A trivial Prolog program

1.3.2 Semantics

The study of programming language semantics deals largely with interpretation, truth and meaning of the program constituents. In the context of logic programming with predicates as basic constituents, every n-ary predicate denotes an n-ary relation over the Herbrand Universe. The semantics of a logic program $\mathcal{P}$ defines a set of n-tuples $< t_1, \ldots, t_n >$ as the denotation (or meaning) of predicate symbol 'p'. Three different kinds of semantics are developed for logic programs: *operational semantics*, *fixpoint semantics* and *model-theoretic semantics*. The denotation of 'p' defined by operational semantics is a set of n-tuples $< t_1, \ldots, t_n >$ such that $p(t_1, \ldots, t_n)$ is provable from $\mathcal{P}$ through a sound inference system.

$$D_o(p) = \{< t_1, \ldots, t_n > | \mathcal{P} \vdash p(t_1, \ldots, t_n)\}$$

Viewing from another angle, a continuous and monotonic transformation $\mathcal{T}_\mathcal{P}$ which maps a set of clauses to ground clauses, is associated with each logic program $\mathcal{P}$. In light of fixpoint semantics, the denotation of 'p' is a set of n-tuples $< t_1, \ldots, t_n >$ such that $p(t_1, \ldots, t_n)$ belongs to the least fixpoint (lfp) of $\mathcal{T}_\mathcal{P}$.

$$D_f(p) = \{< t_1, \ldots, t_n > | \; p(t_1, \ldots, t_n) \in \text{lfp}(\mathcal{T}_\mathcal{P})\}$$

In model theoretic semantics, the denotation of 'p' is a set of n-tuples $< t_1, \ldots, t_n >$ so that $p(t_1, \ldots, t_n)$ is a logical consequence of $\mathcal{P}$.

$$D_m(p) = \{< t_1, \ldots, t_n > | \; \mathcal{P} \models p(t_1, \ldots, t_n)\}$$

These three semantics are shown to be equivalent [vEK76] even though they define seemingly distinct meanings for a given logic program:

Operational	the set of atoms that can be derived from the program
Fixpoint	the least fixpoint of the program transformation operator
Model-Theoretic	the set of atoms that are logically implied by the program

1.3.3 Procedural Interpretation

The informal declarative meaning of a program clause $H \leftarrow B_1, \ldots, B_n$ is "H is implied by B_1 and ... and B_n". It only describes the logical relationships among constituents of a clause. Without procedure interpretation of Horn clauses and an efficient implementation, realization of logic as programming language will be infeasible. Procedural interpretation views the above clause as a procedure definition whose name is H and the body $B_1, \ldots, B_n$ as a set of procedure calls. The procedure H is invoked by a procedure call G_i in the goal statement $\leftarrow G_1, \ldots, G_m$. To execute H, each B_i must be resolved through a series of procedure invocations.

1.3.3.1 Unification

Unification plays an important role in resolution, which is the single inference rule used in (trivial) logic programming. To begin with, the concepts of substitution, unifier and most general unifier are introduced. A substitution is a finite set of the form

$$\delta = \{x_1/s_1, \ldots, x_m/s_m\}$$

where

$x_1, \ldots, x_m$ are distinct variables and each s_i (i=1,...,m) is a term with $x_i \neq s_i$.

Each element x_i/s_i is called the binding for x_i. Composition of two substitutions $\delta = \{x_1/s_1, \ldots, x_m/s_m\}$ and $\psi = \{y_1/t_1, \ldots, y_n/t_n\}$ is the substitution

$$\delta \cdot \psi = \{x_1/s_1\psi, \ldots, x_m/s_m\psi, y_1/t_1, \ldots, y_n/t_n\}$$

with

$x_i/s_i\psi$ being deleted if $x_i = s_i\psi$, and y_j/t_j deleted if $y_j \in \{x_1, \ldots, x_m\}$.

The application of a substitution δ to an atom A, denoted by δA, is the process of replacing all variables of A that appear in δ by their respective bindings. Two atoms A and B are unifiable if there exists a substitution δ such that δA = δB. The substitution δ has a special name called the unifier for A and B (in general, a substitution ψ is called a unifier for a set of atoms S if ψS is singleton). A unifier δ is the most general unifier (MGU) for A and B if there exists a substitution ψ for every unifier ϕ for A and B such that $\phi = \delta \cdot \psi$. The process of finding a (most general) unifier for these atoms is labeled as *unification*.

1.3.3.2 Resolution Principle

Refutation is basically a form of resolution coupled with pattern matching unification. The notion of resolvent is incorporated in both resolution and

refutation [Rob65, CL73]. Given two clauses C_1 and C_2, if there is an atom A in C_1 and its complement B is in C_2, then the resolvent of C_1 and C_2 is

$$\delta[(C_1 - \{A\}) \cup (C_2 - \{B\})]$$

where

δ is the most general unifier of A and B.

A refutation of a set of clauses S is a finite sequence $S_1, \ldots, S_n$ of clauses such that

1. Each S_i $(1 \leq i \leq n)$ is either in S or is a resolvent of any two clauses in $S_1, \ldots, S_{i-1}$.
2. S_n is a null clause.

The null clause is traditionally interpreted as contradiction, which means that S is inconsistent. By the *Resolution Theorem* [Rob65], S is unsatisfiable if and only if there is a refutation of S. This leads to the *Resolution Principle*, which is the root of logic programming.

Theorem 1.1 (Resolution Principle) *Given two clauses C_1 and C_2, a resolvent of C_1 and C_2 is a logical consequence of them.*

1.3.3.3 Model Of Execution

Refutation only states the conditions of a sequence $S_1, \ldots, S_n$ needed to refute a set of clauses S. It doesn't specify what sequence it is or how to obtain such a sequence. A general refutation proof procedure is:

1. Arbitrarily select a subgoal G_i from the goal $G :\leftarrow G_1, \ldots, G_m$ as the next subgoal to solve.

2. Arbitrarily select a clause $C : H \leftarrow B_1, \ldots, B_n$ from the set of program clause $\mathcal{P}$ whose head H is unifiable with G_i.

3. Obtain the most general unifier δ of G_i and H.

4. Replace G_i in G by $B_1, \ldots, B_n$. Apply the unifier δ to it to obtain a new goal $\delta(G_1, \ldots, G_{i-1}, B_1, \ldots, B_n, G_{i+1}, \ldots, G_m)$.

5. Repeat steps 1-4 until either

   (i) The goal is a null clause (i.e. the refutation succeeds) or

   (ii) No clause can be selected in step 2 (i.e. the refutation fails).

This general proof procedure has dual functionalities. It possesses the ability to prove the inconsistency of a set of clauses S, and to find the variable substitution leading to a refutation. There are many refutation procedures rested upon different refinements of Resolution Principle. One such method

is *SL-Resolution for Definite Clauses* or *SLD-resolution.* SL stands for *Linear Resolution with Selected Function.* The selected function, also known as the *Computation Rule*, imposes a restriction on *how* to select the next subgoal (in step 1). The value of this function for a goal clause is an atom named selected atom of the goal clause. The refutation process can also be visualized as an AND/OR search tree:

1. The root is the initial goal statement $\leftarrow G_1, \ldots, G_m$.
2. An OR-branch, which links an atom with the head of a clause, is constructed for each clause $H \leftarrow B_1, \ldots, B_n$ whose head $H$is unifiable with the selected subgoal G_i.
3. An AND-branch is constructed to connect the body $B_1, \ldots, B_n$ of a clause with its corresponding head H.

Compiling with procedural interpretation of Horn clauses, an OR-branch depicts the unification between a subgoal G_i and the head of an applicable procedure H, an AND-branch manifests the execution of procedure H through a series of procedure invocations designated by the conjunction of its body B_1 and ... and B_n. Fig 1.2 illustrates the AND/OR proof tree for executing the 'grandson' program in Fig 1.1

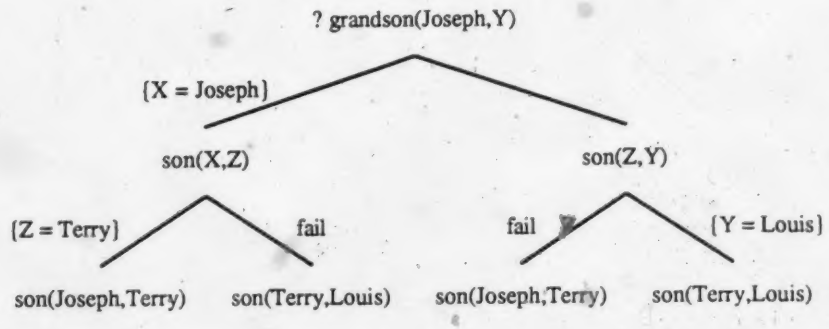


Figure 1.2: AND/OR proof tree of a logic program execution

Chapter 2

Functional Logic Languages

Although logic programming and functional programming are basically declarative in nature, there are fundamental differences between them. Despite their essential and stylistic discrepancies, each of them provides many special and complementary features which makes it worthwhile to integrate them under one roof. This section outlines and compares their characteristics, and provides some general ideas of combining these two paradigms into a unified framework.

2.1 Logic versus Functional Programming

A program is non-deterministic if it admits more than a single legitimate computation resulting in (possibly) multiple solutions. Pure Horn clauses

logic programs are inherently non-deterministic. Visualizing the computation as an AND/OR search tree, it has more than one branch (either AND-branch or OR-branch) originating from a tree node. Such branching has two contributing factors:

- the body of a goal is considered as a set of atoms, and the order of procedure invocations, designated by these atoms, can be arbitrary (AND branch).
- arbitrary selection of candidate clause whose head is unifiable with a given goal (OR branch).

SLD-resolution, which imposes a computation rule for subgoal selection, does not prune the non-deterministic characteristic of logic programs. Non-deterministic logic languages reveal the capability of separating logic from its control counterpart. This exclusive feature admits opportunities for parallel execution. Two forms of parallelism, *OR-parallelism* and *AND-parallelism*, are commonly exploited. One reason of non-determinism in logic languages is the possible application of several candidate clauses to solve a single atom (of a goal). OR-parallelism is simply the concurrent invocation of all these clauses. AND-parallelism exploits the fact that conjunction of procedure calls (in a goal) can be executed in parallel.

Similar to the case that no restriction is imposed on the execution order, pure logic languages also make no explicit commitment as to the input and

output of a clause. Variable substitution extracted by unification can be divided into two parts: input substitution contains a set of bindings for variables appearing in the clause head, and output substitution contains a set of bindings for variables appearing in the goal. Unlike imperative languages, which delimit input and output parameters of a procedure, input and output substitutions are not fixed (even for the same clause). Their compositions are determined by both a goal and the clause head, and variable bindings are generated without regard of the origin of the variables. Logic languages are trivially functional invertible as a result of the bi-directional data flow.

During resolution, non-ground terms can be supplied as input to a clause or obtained as output binding. Variables appearing in these non-ground terms can be partially instantiated by a predicate application and later on further instantiated by other predicate applications. Variables of this nature are called logical variables. Data structure containing logical variables, also known as partial data structure, can now be used to represent possibly infinite data structure without explicitly identifying all the elements.

Computation in functional languages consists in the evaluation of a term to its corresponding value defined by a sequence of function applications. Equivalence between a term and its value reinforces the concept that functional programming can be based on first-order equational logic [CM79, HO82, BG86] and suggests the use of equations in defining a functional program. A functional program is typically written in the form of equations

used as left-to-right rewrite rules. The evaluation mechanism is reduction, which matches an expression against the left hand side of an equation, obtains bindings for variables in the equation, and replaces the expression by corresponding right hand side of the equation. The rewriting process continues until the expression is irreducible (i.e. contains only constants or constructors). Reduction differs from unification in that variable bindings are only generated for those variables appear in the matched equation, which implies unidirectional flow of information from the expression to an equation. Coping with mathematical definition of function, functional program denotes proper many-to-one mapping. A functional program is deterministic in the sense that it only admits a single answer for each function evaluation. Functions in functional program are treated in the same way as partial data structures are treated in logic program. As first class objects, they can be passed as parameters of functions or returned as values of function applications.

Benefited from the well-known Church-Rosser property, repeatly rewriting the leftmost redex of an expression always leads to a normal form (if it really exists) [GHT84]. Indeed, searching is unnecessary in the evaluation of functional programs. Absence of exhaustive navigation simplifies computation complexity and enables flexible runtime control (such as parallel execution). In contrast, a search process is mandatory in proving a goal from a set of clauses. The runtime behavior of non-deterministic

logic programs is far more complicated and more difficult to control. Since logic languages are functional invertible (as a result of non-directional input-output), a program can be specified without any control information. The lack of control information leads to certain undesirable runtime behaviors such as redundant computation and non-terminating evaluation. However, this very same factor contributes to the enhancement of the expressive power of logic languages. Coupling with the capacity to handle partially instantiated data structures, a single logic program can perform the same task as several functional programs. The situation is further cokmplicated by the fact that logic programming is inherently first-order. All function symbols are then considered 'flat' in nature. They are only used as constructors for data structures rather than as defined function applications. Treating functions as first class object, functional languages possess the ability to define higher-order functions. In analogue to abstract data type in imperative languages, related functions are now realized as a specific instance of a generic function. The expressive power of functional languages is, on this count, greater than typical logic languages.

Life will be much easier if any language is definitely superior to the others. That is not the case for functional and logic languages. Their similarities and discrepancies are best explained by the fact that they employ different operational semantics on top of comparable logical frameworks. Horn clause logic programming uses a restricted subset of first-order predicate

logic, whereas functional programming uses an equational form of logic. Function evaluation is carried out through pattern matching reduction, which treats equations defined by a functional program as left-to-right rewrite rules. Since predicates can be rewritten as functions, it appears at first sight that functional languages should be as expressive as logic languages. This viewpoint is shadowed by the fact that function definition is a function symbol attributed with a desirable 'interpretation'. In addition, unification-based resolution employed by logic languages subsumes pattern matching reduction, making logic languages more expressive than functional languages. Taking advantage of their declarative nature and all these programming features, amalgamation of logic and functional languages can be achieved through integration of their underlying logic and discovery of a suitable operational semantics for the unified framework.

2.2 Operational Semantics

The technique of adding an equational flavor to Horn clause logic is widely used in amalgamating functional and logic languages. The origin of their discrepancies is partially addressed to the use of different operational semantics. Procedural semantics of logic languages is unification-based resolution, whereas the evaluation mechanism of functional languages is pattern matching reduction. Narrowing, which combines unification with

term rewriting, serves as the bridge of integration. This section outlines the notation used throughout the thesis and provides some insights of reduction and narrowing.

2.2.1 Preliminaries

The well-known notations and concept of algebra are adopted from [Hul80, HO80]. The set of terms composed from a set of function symbols $\mathcal{F}$ and a set of variables $\mathcal{V}$ is labeled as $\mathcal{T}(\mathcal{F}, \mathcal{V})$. The subterms within a given term are formally referenced by their occurrences. An occurrence is a (possibly empty) sequence of positive integers, indicating the relevant position (from the outermost principal functor) of a subterm. The set of occurrences for a term A, denoted by $\mathcal{O}(A)$, is defined as a finite subset of Z^* where

1. $\xi \in \mathcal{O}(A)$

2. $u \in \mathcal{O}(A_i) \quad \Leftrightarrow \quad i \cdot u \in \mathcal{O}(f(A_1, \ldots, A_n)) \qquad (1 \leq i \leq n)$

ξ is the empty sequence and $i \cdot u$ is the concatenation of i and u. For example, the set of occurrence for the term f(g(X),Y) is $\{\xi, 1, 1 \cdot 1, 2\}$, and the subterm g(X) is addressed by occurrence $u = 1$. $\mathcal{O}(A)$ is partially ordered by the prefix ordering: (i) $u \leq v$ iff there exists a 'w' such that $v = u \cdot w$ (i.e. $w = u - v$) (ii) u and v are disjoint iff $u \not\leq v$ and $v \not\leq u$. The subterm of A at occurrence u, denoted by A/u is

1. A if $u = \xi$

2. A_i/v if A is of the form $f(A_1, \ldots, A_n)$ and $u = i \cdot v \quad (1 \leq i \leq n)$

The set of variables appear in A is denoted by $\mathcal{V}(A)$ and the non-variable subset of $\mathcal{O}(A)$ is denoted by $\bar{\mathcal{O}}(A)$. If $u \in \mathcal{O}(A)$, then the resulting term of replacing the subterm A/u by another term B, written as $A[u \leftarrow B]$ is

1. B if $u = \xi$

2. $f(A_1, \ldots, A_i[v \leftarrow B], \ldots, A_n)$ if $A = f(A_1, \ldots, A_n)$ and $u = i \cdot v$

Recall that a substitution is a mapping from $\mathcal{V}$ to $\mathcal{T}(\mathcal{F}, \mathcal{V})$. For every substitution δ, the set of variables affected by δ is $\mathcal{D}(\delta) = \{x \mid \delta(x) \neq x\}$ and the set of variables introduced by δ is $\mathcal{I}(\delta) = \bigcup_{x \in \mathcal{D}(\delta)} \mathcal{V}(\delta(x))$. The restriction of δ to a subset of variables $\mathcal{W}$, denoted by $\delta \uparrow \mathcal{W}$, is defined as

$$\begin{cases} (\delta \uparrow \mathcal{W})(x) = \delta(x) & \textit{if } x \in \mathcal{W} \\ (\delta \uparrow \mathcal{W})(x) = x & \textit{otherwise} \end{cases}$$

Intuitively, variable substitution application will only affect the variables designated by $\mathcal{W}$. An equational theory E is a finite set of equations, whereas the E-equality ($=_E$) is the finest congruence closed under instantiation and generated by E. For any equation $A = B$ in E and any arbitrary substitution δ, $=_E$ contains all pairs $< \delta A, \delta B >$ such that $\delta A = \delta B$.

Definition 2.1 *Extending E-equality to substitution,* $\delta =_E \theta$ *iff for all variable* x $\delta(x) =_E \theta(x)$.

Definition 2.2 *A substitution* δ *is more general than* θ *under E, denoted by* $\delta \leq_E \theta$, *iff there exists another substitution* ψ *such that* $\psi \cdot \delta =_E \theta$.

Let A and B be two terms, $\mathcal{V} = \mathcal{V}(A) \cup \mathcal{V}(B)$ be the set of variables that appear in these two terms, $\mathcal{W}$ be a finite set of variables containing $\mathcal{V}$ and $\bigcup_E(A, B)$ be the set of all E-unifiers of A and B. A set of substitutions Σ is a complete set of unifiers of A and B away from $\mathcal{W}$ iff

1. $\forall \delta \in \Sigma \quad \mathcal{D}(\delta) \subseteq \mathcal{V}$ and $\mathcal{I}(\delta) \cap \mathcal{W} = \emptyset$
2. $\Sigma \subseteq \bigcup_E(A, B)$
3. $\forall \delta \in \bigcup_E(A, B) \quad \exists \theta \in \Sigma$ such that $(\theta \leq_E \delta) \upharpoonright \mathcal{W}$

These basic restrictions require all substitutions in Σ **must** be an E-unifier of A and B; and for every unifier δ of A and B, there exists an unifier θ in Σ which is more general than δ. Σ is said to be minimal if it satisfies the further condition that no two unifiers in Σ are comparable,

4. $\forall \delta, \theta \in \Sigma \quad \delta \neq \theta \Rightarrow (\delta \not\leq_E \delta) \upharpoonright \mathcal{W}$

2.2.2 Reduction

Reduction is an evaluation process which matches an expression against the left hand side of a rewrite rule, obtains variable bindings for the rule and replaces the expression by corresponding right hand side of the rule. Each rewrite rule is a directed equation of the form

$$\alpha_k \rightarrow \beta_k$$

with no extra variable on the right hand side (i.e. $\mathcal{V}(\beta_k) \subseteq \mathcal{V}(\alpha_k)$). The set of rewrite rules defines a term rewriting system $\mathcal{R}$.

Definition 2.3 *A term A is reduced to another term B at a non-variable occurrence u, written as $A \rightarrow_{[u,k]} B$, iff there exists a rewrite rule $\alpha_k \rightarrow \beta_k$ in $\mathcal{R}$ and a substitution δ for α_k such that $A/u = \delta(\alpha_k)$ and $B = A[u \leftarrow \delta(\beta_k)]$*

$A \rightarrow_{[u,k]} B$ is also written as $A \rightarrow B$ if no confusion arises. A sequence of reduction $A \rightarrow_{[u_0,k_0]} A_1 \rightarrow \cdots \rightarrow_{[u_{n-1},k_{n-1}]} B$ is called a reduction derivation. It is also written as $A \stackrel{*}{\rightarrow}_{[U,K]} B$ where [U,K] denotes the sequence of $[u_i, k_i]$. The one-step reduction relation $\rightarrow_{\mathcal{R}}$ associated with $\mathcal{R}$ is the relation over $\mathcal{T}(\mathcal{F}, \mathcal{V})$ which contains $\mathcal{R}$ and is closed by substitution and replacement. The transitive-reflexive closure of $\rightarrow_{\mathcal{R}}$ is denoted by $\stackrel{*}{\rightarrow}_{\mathcal{R}}$. Note that $\stackrel{*}{\rightarrow}_{\mathcal{R}}$

is not the same as reduction derivation. There could be more than one derivation from A to B for any relation $A \stackrel{*}{\rightarrow}_{\mathcal{R}} B$. The equational theory of $\mathcal{R}$ is obtained by replacing '=' with '$\rightarrow$'.

Eg. 2.1.,

Given a T.R.S.,

$a \rightarrow b$

$a \rightarrow c$

$b \rightarrow d$

$c \rightarrow d$

The relation $a \stackrel{*}{\rightarrow}_{\mathcal{R}} d$ can be achieved through either $a \rightarrow b \rightarrow d$ or $a \rightarrow c \rightarrow d$

A term A is in $\rightarrow_{\mathcal{R}}$ normal form iff it cannot be further reduced in $\mathcal{R}$. A substitution is normalized if all of its substitutes are in normal forms. A term can be reduced in many different ways Depending on which subterm is being rewritten and which rewrite rule is selected, the $\rightarrow_{\mathcal{R}}$ normal form (if exists) may not be unique. A reduction relation $\rightarrow_{\mathcal{R}}$ is

Noetherian if there is no infinite reduction sequence for all the terms

Confluent if for all terms A, B, C $A \stackrel{*}{\rightarrow}_{\mathcal{R}} B$ and $A \stackrel{*}{\rightarrow}_{\mathcal{R}} C$ iff there exists a term D such that $B \stackrel{*}{\rightarrow}_{\mathcal{R}} D$ and $C \stackrel{*}{\rightarrow}_{\mathcal{R}} D$

A term rewriting system $\mathcal{R}$ is noetherian (confluent) iff $\rightarrow_{\mathcal{R}}$ is noetherian (confluent). $\mathcal{R}$ is canonical iff it is confluent and noetherian. Such system admits a unique $\rightarrow_{\mathcal{R}}$ normal form for all the terms. This property facilitates a decision procedure for equational theories as two terms are equal iff they have the same $\rightarrow_{\mathcal{R}}$ normal formal.

2.2.3 Narrowing

The basic idea of narrowing is to combine unification with term rewriting. Informally, narrowing a term is applying to it the minimum substitution such that it is reducible and then reduce it.

Definition 2.4 *A term A is narrowed to another term B at a non-variable occurrence u, written $A \rightsquigarrow_{[u,k,\delta]} B$, iff there exists a rewrite rule $\alpha_k \rightarrow \beta_k$ in $\mathcal{R}$ and a most general unifier δ for A/u and α_k such that $\delta(A/u) = \delta(\alpha_k)$ and $B = \delta(A[u \leftarrow \beta_k])$.*

A sequence of narrowing $A \rightsquigarrow_{[u_0,k_0,\rho_0]} A_1 \rightsquigarrow \cdots \rightsquigarrow_{[u_{n-1},k_{n-1},\rho_{n-1}]} B$ is called a narrowing derivation. It is also written as $A \stackrel{*}{\rightsquigarrow}_{[U,K,\rho]} B$ where $[U, K, \rho]$ denotes the sequence of $[u_i, k_i, \rho_i]$ and $\rho = \rho_{n-1} \cdots \rho_0$. The notion of $\rightsquigarrow_{\mathcal{R}}$, $\stackrel{*}{\rightsquigarrow}_{\mathcal{R}}$

and $\rightsquigarrow_{\mathcal{R}}$ normal forms are analogous to those of reduction. Narrowing differs from reduction in two major aspects: (i) existence of more than one $\rightsquigarrow_{\mathcal{R}}$ normal form is possible (even for canonical rewriting systems) (ii) narrowing can be used to solve the variables of a non-ground expression.

Given a canonical term rewriting system, every term must have an unique normal form [HO80]. The relationship between a term and its normal form is traditionally a many-to-one mapping instead of a more restrictive one-to-one mapping. A simple proof procedure is made possible through this property: two terms are equal iff they can be reduced to an identical canonical normal form. Narrowing is essentially a hybrid inference mechanism combining reduction with unification. Their characteristics and inter-relationship are well-established in Hullot's landmark paper [Hul80]. Under the same canonical setting, a narrowing derivation is associated with each reduction derivation or vice versa. For any reduction derivation of

$$\delta(A) \stackrel{*}{\rightarrow}_{[U,K]} A_n$$

with δ being a normalized substitution, there exists a corresponding narrowing derivation

$$A = B_0 \stackrel{*}{\rightsquigarrow}_{[U,K,\rho]} B_n$$

where γ is a normalized substitution such that $\gamma B_n = A_n$ and $\gamma\rho \leq_E \delta$

The ultimate goal of unification is to construct a (most general) unifier for two terms A and B. In parallel with unification, narrowing can also be used

to obtain the E-unifier of two terms, A and B, modulo a given equational theory E through repetitive narrowing of A and B. Another elegant result from [Hul80] is the completeness characteristic of narrowing for a canonical system. A complete set of E-unifiers for any two unifiable terms exists, and each such substitution can be enumerated through narrowing.

2.3 Narrowing Strategies

In the context of term rewriting systems, the intended semantics of a function symbol can be specified by more than one rewrite rule. Even if each rule defines a single function, a term can be narrowed at different occurrences using (possibly) different rules. It can have more than one normal form depending on which subterm is narrowed and which rewrite rule is applied. This highly non-deterministic property is costly, unfavorable and often leads to redundant computation of comparable solutions. A number of narrowing strategies have been employed to rectify this undesirable characteristic. This section presents the fundamental notions of two different strategies, innermost narrowing and outermost narrowing [DV87]. It is shown that using any strategy alone is inadequate in the sense that complete set of minimal unifiers cannot be enumerated effectively.

When a term is rewritten, new subterms may be introduced through instantiation. One source of redundancy is addressed to subsequent

narrowing of these newly instantiated subterms. Hullot applied the concept of *basic narrowing* [Hul80] to redress the problem. The idea of basic narrowing is to rewrite non-variable occurrences of the original term only. Given a term A, each narrowing derivation is based on $\bar{\mathcal{O}}(A)$ and narrowing step on the introduced subterms is avoided. Using basic narrowing as a stepping stone, Rety extended it to *normalized narrowing* [Ret87](i.e. normalize the term after each narrowing step) which computes the basic occurrences while preserving the completeness.

Informally, *innermost narrowing* selects an innermost occurrence of a term as the next candidate to rewrite. Given a term A, a subterm at occurrence $u \in \bar{\mathcal{O}}(A)$ is innermost if all subterms at occurrences $v > u$ are in normal form. Intuitively, innermost narrowing of a term $f(t_1, \ldots, t_n)$ means that the function 'f' is applied to the corresponding normal forms of all the subterms $t_1, \ldots, t_n$. Using the terminology of imperative programming, innermost narrowing is the analogy of pass by value procedure invocation. The term $f(t_1, \ldots, t_n)$ is treated as a procedure invocation in which all the parameters $t_1, \ldots, t_n$ are evaluated before the procedure 'f' is executed. Although repetitive rewriting of identical subterms is avoided, it is generally undesirable to evaluate all the subterms before the function is applied (especially for a more general non-terminating rewriting system). Consider a trivial definition of the if-then-else statement: $\{\text{if}(\text{true}, X, Y) \rightarrow X, \text{if}(\text{false}, X, Y) \rightarrow Y\}$. The intention of these rules is to

rewrite either X or Y (but not both) pending the truth value of the first argument. Narrowing both of them opens the door for potential problems such as unfruitful computation and non-terminating derivation.

Eg. 2.2.,

Given a T.R.S.,

$$\text{if}(\text{true}, X, Y) \rightarrow X$$
$$\text{if}(\text{false}, X, Y) \rightarrow Y$$
$$f(0) \rightarrow f(f(0))$$

Narrowing a term if(true, 0, f(0)) at the innermost occurence leads to infinite derivation of the subterm f(0). Notice that this derivation does not even contribute to the narrowing of the original term (i.e. it is an unfruitful computation).

Outermost narrowing handles the problem in a different manner. It selects an outermost occurrence as the next candidate to rewrite. Given a term A, a subterm at occurrence $u \in \bar{\mathcal{O}}(A)$ is outermost if it is narrowable and for all narrowable occurrences $v \in \bar{\mathcal{O}}(A)$ either $u \leq v$ or $u <> v$. The subterms of $f(t_1, \ldots, t_n)$ are narrowed when necessary after the function 'f' is applied. Similar to call-by-name procedure invocation, the term

$f(t_1,\ldots,t_n)$ is 'simplified' when possible regardless the 'value' of all the subterms. Consider the previous example, the term if(true, 0, f(0)) is now correctly narrowed to '0' without evaluating f(0). Notice that if there is no unfruitful and infinite innermost derivation, innermost narrowing is in general more efficient than outermost narrowing because it avoids narrowing identical copies of the subterms. Despite its capability to get around unnecessary computation, outermost narrowing is an incomplete strategy. More precise, outermost or innermost narrowing alone may not generate complete set of unifiers for two unifiable terms. As noted by Fribourg [Fri85], conventional function definitions are 'incompleted' and denote 'partial functions'.

Eg. 2.3.,

Given a T.R.S.,

$$\begin{aligned} &\mathrm{multi}(\mathrm{pred}(0),0) \rightarrow 0 \\ &\mathrm{multi}(X,1) \rightarrow X \\ &\mathrm{pred}(1) \rightarrow 0 \end{aligned}$$

Unifying multi(pred(X),X) with 0, innermost narrowing leads to

$$\mathrm{multi}(\mathrm{pred}(X),X) \leadsto_{\{X/1\}} \mathrm{multi}(0,1) \leadsto 0$$

while outermost narrowing leads to

$$multi(pred(X), X) \rightsquigarrow_{\{X/0\}} 0$$

Uncomparable results, $\{X/1\}$ and $\{X/0\}$, are generated which signifies that employing innermost or outermost narrowing alone is inadequate.

Chapter 3

E_C-Equality Theory

3.1 Equality Theory

The relationship between different terms is conventionally established through a set of equations. Informally, the left hand side of an equation is considered 'semantically the same' as its corresponding right hand side. These equations collectively define an equational theory E. The piece of puzzle that remains to be solved is to prove the equality between different terms within a well-defined framework. The proof is traditionally confined to classical equality theory described by the axioms:

Reflexivity $\forall x \;\; x = x$

Symmetry $\forall xy \;\; x = y \rightarrow y = x$

Transitivity $\forall xyz \ (x = y) \wedge (y = z) \rightarrow x = z$

Substitutivity $\forall x_1 \ldots x_n \forall y_1 \ldots y_n \quad (x_1 = y_1) \wedge \ldots \wedge (x_n = y_n) \rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$ for all function symbols 'f'.

The theoretical formalism of classic equality is particularly difficult to handle despite its simplistic notation and elegant semantics. The success of Prolog-like logic languages is largely accredited to the avoidance of equality, resulting in an efficient implementation of the underlying operational semantics. A key to combine logic and functional languages is to overcome this 'flat' nature of Prolog-like languages. Ordinary syntactic unification is modified to accommodate the semantics of defined function symbols. It is generally preferable to employ equational reasoning in *extended unification*. E-unification [Plo72] provides a basis to incorporate equality into resolution. Equation solving can now be conceived as a notion of unification modulo an equational theory. The observation that functional languages can be formalized by equational first-order logic suggests a generalized method to unify logic and functional languages by readmitting equality to SLD-resolution [vEL84, vEY87]: transform some initial goals to a set of equations and solve the equations instead. Program execution is now composed of two different stages: equation generation and equation solving. Given a logic program, the first stage is to construct an SLD-derivation from some initial

goals to a set of equations E′. i.e. to show that

$$E \cup Eq \models \exists E'$$

where E is an equational theory defined by a set of equations and Eq is an equality theory. The second phase is to prove consistency of the newly generated set of equations by means of SLD-resolution. It is normally carried out by resolving the set of equations with those in E and Eq. Substitutions for the variables, δ in the initial goals are produced to prove that

$$E \cup Eq \models \forall\delta(E')$$

This approach is a generalization of (extended) unification. Ordinary unification is obtained if the equational theory E is empty and the equality theory Eq is $\{\forall x \;\; x = x\}$. If Eq is the standard equality theory and E is non-empty, then the inference system is one of the extended unification [Kor83, GM84, DV87, LPB$^+$87]. Taking to the extreme, Eq can be any theory specifying relations other than equality. It becomes one of the constraint solving languages [JL87] when augmented with a suitable 'solving' system.

3.2 E_c-Equality

Until recently, equality theory employed in extended unification is restricted to classic equality. Taking into consideration the intended semantics of functional programming, variants of classic equality theory have been proposed hoping that a practical and efficient manipulation of such equality may exist. One stream is to distinguish function applications from the constructor symbols that are used as data structures [Fri85, O'D85, LPB+87]. This section discribes a variation of equality based on the division of functional symbols, named E_c-equality. This subset of classic equality has a distinct characteristic in which complete set of minimal E_c-unifiers always exists for a constructor-based term rewriting system.

Given a term rewriting system $\mathcal{R}$, the set of function symbols $\mathcal{F}$ is classified into two disjointed categories: defined function symbols $\mathcal{F}_d$ and constructors $\mathcal{F}_c$. A defined function symbol is the outermost principal functor appearing on the left hand side of a rule and constructors are the remaining functors used to build the data structures.

Eg. 3.1.,

Given a T.R.S.,

$$f(h(X)) \rightarrow g(X)$$
$$g(X) \rightarrow l(X)$$

The defined function symbols are 'g' and 'f', while the constructor symbols are 'h' and 'l'.

A constructor term is either a constructor or a term composed of constructors and variables only. The set of constructor terms is denoted by $\mathcal{T}(\mathcal{F}_c, \mathcal{V})$. A term that contains at least one defined function symbol is called a function term. The equality relation is defined as a subset of classic equality which captures the semantic notion that function applications on constructor terms are equaled to their values.

Definition 3.1 *Two terms A and B are E_c-equivalent, denoted $A =_{E_c} B$ iff there exists a constructor term C such that $A =_E C$ and $B =_E C$*

For any confluent rewriting system, $=_E$ is the symmetric closure of $\stackrel{*}{\rightarrow}$, and the Church-Rosser property suggests that $A \stackrel{*}{\rightarrow} C$ and $B \stackrel{*}{\rightarrow} C$. Hence, proof procedures based on reduction can also be used to prove E_c-equality. Two terms are E_c-equivalent iff there they reduce to a common constructor term. A term A may even not E_c-equivalent to itself if A cannot be reduced to a constructor term or it is trapped in an infinite derivation. Analogous to E-equality, two terms A and B are E_c-unifiable iff there exists a E_c-unifier δ such that $\delta A =_{E_c} \delta B$. Differs from [Fri85], an E_c-unifier may contain function

terms rather than compose of constructor terms only. Comparison of E_c-unifiers are the same as in E-equality. Obviously, $=_{E_c}$ is a subset of $=_E$ in which each congruence class must contain a (unique) constructor term. Despite the elegant semantics of E_c-equality, a complete set of minimal E_c-unifiers may not exist for a general rewriting system $\mathcal{R}$.

Eg. 3.2.,

Given a T.R.S.,

$$g(X, Y) \rightarrow X$$
$$f(g(X, Y)) \rightarrow f(X)$$
$$f(a) \rightarrow b$$

Infinite number of E_c-unifiers exist for f(Z) and f(a).

$$\delta_0 = \{Z/a\}$$
$$\delta_1 = \{Z/g(a, Y_1)\}$$
$$\ldots\ldots$$
$$\delta_n = \{Z/g(g(\ldots g(g(a, Y_n), Y_{n-1})\ldots), Y_1)\}$$
$$\ldots\ldots$$

It is obvious that each δ_i is a E_c-unifier of $f(\bar{Z})$ and f(a). For each consecutive pair of unifiers δ_{i-1} and δ_i, there exists an arbitrary

substitution τ for Y_i such that $\tau\delta_i \uparrow \{Z\} =_E \delta_{i-1} \uparrow \{Z\}$. In other words, δ_i is more general than the preceding unifier δ_{i-1}. Thus, complete and minimal set of E_c-unifiers does not exist for such a system.

Notice that in this example

$$X =_E g(X, Y_1) =_E \ \ldots \ =_E g(g(\ldots g(g(X, Y_n), Y_{n-1})\ldots), Y_1) \ =_E \ldots$$

and the left hand side of the rule $f(g(X, Y)) \rightarrow f(X)$ is narrowable at occurrence of u = 1 (i.e. at the function symbol 'g'). Hence,

$$f(Z) =_E f(g(Z, Y_1))_E \ \ldots \ =_E f(g(g(\ldots g(g(Z, Y_n), Y_{n-1})\ldots), Y_1)) \ =_E \ \ldots$$

with each Y_i being unspecified, making it impossible to obtain the 'minimal' unifier for f(Z) and f(a). The problem is sufficiently resolved when no different nonvariable part of the left hand side of any rule may have a common instance. It is also known as nonambiguous or superposition free in the literature.

Definition 3.2 *A term rewriting system is left linear if no variable appears more than once in the left hand side of any rule*

Definition 3.3 *A term rewriting system $\mathcal{R}$ is non-overlapping if there is no*

critical pair in $\mathcal{R}$ (i.e. for any two rules $\alpha_i \rightarrow \beta_i$ and $\alpha_j \rightarrow \beta_j$, α_i/u and α_j $(u \in \bar{\mathcal{O}}(\alpha_i))$ have no common instance except when $u = \xi$ and $i = j$)

These two properties are syntactically checkable and guarantee the confluence property without resorting to the termination property. Non-existence of 'minimal' E_c-unifier is lifted if $\mathcal{R}$ is further restricted to a constructor-based term rewriting system. In [Hul80], the stepping stone for any complete proof procedure is an unique canonical normal form of a term, which may not exist in $\mathcal{R}$. Although $\mathcal{R}$ may be non-terminating, a similar procedure is made possible under E_c-equality. Equivalence between 'undefined' terms are effectively purged since these terms are not equaled to any single term at all.

Definition 3.4 *A term rewriting system is constructor-based if it is left-linear, non-overlapping and no defined symbols appearing in the inner part of the left hand side of any rule.*

3.3 Complete set of Minimal E_c-unifiers

This section presents completeness and minimality characteristics of E_c-unifiers for equational theory described by a constructor-based term rewriting system. More precisely, a complete set of minimal E_c-unifiers always exists

for such system. A slightly more general notion of normal form, named S-normal form, is introduced in [You89]. The set of S-normal forms includes all irreducible function terms in addition to the set of constructor terms. Two terms are E_n-equivalent iff they are both E_n-equivalent to a common S-normal form. A fundamental difference between E_n-equality and E_c-equality lies in the treatment of undefined (or irreducible) function terms. An irreducible function term is still E_n-equivalent to itself. Indeed, $=_{E_c}$ is a subset of $=_{E_n}$. Completeness results holding for E_n-equality should also be valid for the slightly more restrictive setting of E_c-equlaity. The theorems and lemmas presented in the following sequels are adapted from [You89] and refer to the literature for their proofs. For technical purpose, assume 'H' is a new constructor not in $\mathcal{F}$.

Theorem 3.5 *For any E_c-unifier of τ A and B, there exists a narrowing derivation*

$$H(A, B) \overset{*}{\leadsto}_{[U,K,\rho]} H(C_1, C_2)$$

where δ is the most general unifier of the constructor terms C_1 and C_2, and τ is an E_c-unifier of A and B so that $\delta\rho \leq_E \tau \uparrow \mathcal{V}$

The theorem states that for any E_c-unifier, a more general E_c-unifier can be obtained from narrowing. The complete set of E_c-unifiers Σ for A and B can

be enumerated through all narrowing derivations originated from H(A,B). Consider a narrowing derivation of

$$\mathrm{H}(A, B) \leadsto_{[u_0, k_0, \rho_0]} \cdots \leadsto_{[u_{n-1}, k_{n-1}, \rho_{n-1}]} \mathrm{H}(C_1, C_2)$$

where δ is the most general unifier of C_1 and C_2. Since $\mathcal{R}$ is left-linear, every substitute in any $\rho_i \uparrow \mathcal{V}$ must be extracted from the left hand side of a rule. By the non-overlapping property of $\mathcal{R}$, none of these substitute unify with the head of any rule. In other words, all the substitutes ρ_i and δ are normalized. For each E_c-unifier in Σ, all substitutes obtained as such are also normalized. Two unifiers are comparable by $\leq_E$ iff they are comparable by $\leq$ (for example, f(Y) $\leq$ f(g(X))). Complete set of E_c-unifiers is best manifested by a digraph with distinct components. Each distinct sub-digraph (component) represents the $\leq$ ordering of comparable solutions. A unique lower bound (subject to variable renaming) always exists for any such ordering. Uncomparable unifiers are trivially denoted by single-noded sub-digraphs. Hence, the set of all lower bounds plus all the uncomparable E_c-unifiers (single-noded sub-digraph) form a complete and minimal set. An example of digraph representation of complete set of minimal E_c-unifiers is demonstrated in Fig 3.1.

Theorem 3.6 *Complete and minimal sets of E_c-unifiers always exist for any E_c-unifiable terms.*

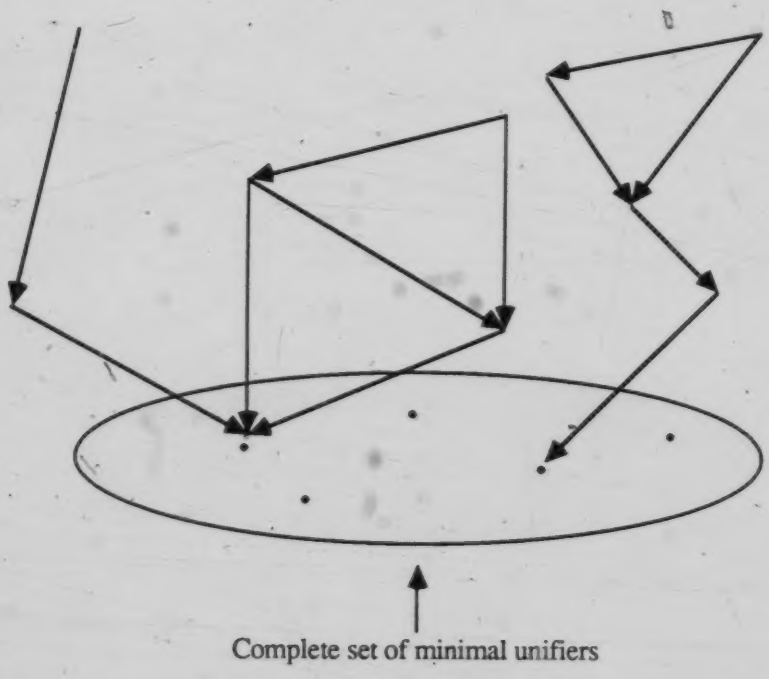


Figure 3.1: Digraph of complete and minimal set of E_c-unifiers

The theorem only claims the existence, not finiteness, of complete sets of minimal E_c-unifier.

Eg. 3.3.,

Given a T.R.S.,

$$f(c(X)) \rightarrow f(X)$$
$$f(d) \rightarrow e$$

An infinite number of uncomparable E_c-unifiers $\{X/d\}$, $\{X/c(d)\}$, $\{X/c(c(d))\}$, ... exists for $f(X)$ and e.

3.4 Outer-narrowing

Two examples in the previous chapter (Eg. 2.2 and Eg. 2.3) suggest the need of an efficient means to interleave innermost and outermost narrowing: *if an inner rewriting is subsequently followed by an outer rewriting, then it* **must** *contribute in certain ways to the reducibility of that outer rewriting.* Inspired by this principle, an *outer-narrowing* derivation is defined as a sequence of narrowing steps such that any later narrowing step at some outer occurrence cannot be preformed earlier using the same rule at the same occurrence. This essential requirement is the Outer-Before-Inner property of any outer-narrowing derivation.

Eg. 3.4.,

Given a T.R.S.,

$$\text{add}(X, 1) \rightarrow \text{succ}(X)$$

$$\text{pred}(1) \rightarrow 0$$

$$\text{succ}(0) \rightarrow 1$$

The narrowing sequence of

$$\text{add}(\text{pred}(X), Y) \leadsto_{\{X/1\}} \text{add}(0, Y) \leadsto_{\{Y/1\}} \text{succ}(0) \leadsto 1$$

does not constitute an outer-narrowing derivation as the second narrowing step involving the rule $\text{add}(X, 1) \rightarrow \text{succ}(X)$ can be carried out right at the very beginning. However, it can be rearranged to a similar outer-narrowing derivation

$$\text{add}(\text{pred}(X), Y) \leadsto_{\{Y/1\}} \text{succ}(\text{pred}(X)) \leadsto_{\{X/1\}} \text{succ}(0) \leadsto 1$$

which generates the same solution.

Conceptually, all narrowing derivations for a given term are partitioned into disjointed equivalent classes. Two derivations are in the same class if they generate comparable solutions. In other words, for any two solutions (say δ and τ) so generated in these two derivations, either $\tau \leq_E \delta$ or $\delta \leq_E \tau$. The

idea behind outer-narrowing is that an unique outer-narrowing derivation, which produces the most general substitution, exists for each class. All other narrowing derivations within the same class can then be 'rearranged' to the outer-narrowing derivation that represents the entire class. . Complete set of unifiers is computed by enumerating all outer-narrowing sequences which are the proxy for different classes of narrowing derivation. Left-linearity of constructor-based rewriting system guarantees solo appearance of any variable on the left hand side of a rule; No such restriction is imposed on the corresponding right hand side. Different 'copies' of a subterm are created when it unifies with a variable on the left hand side and 'distributed' to the same variable on the right hand side of the same rule. The terminology of residue map is introduced to keep track of all these 'copies' of a subterm for a precise description of the rearranging procedure. Note that the residue map is a set of residues since identical copies of a subterm can be created.

Definition 3.7 *If a term A is reduced to another term B at occurrence u using the k^{th} rule, then the residue map for reduction with respect to the rewriting system $\mathcal{R}$ is defined as follows.*

$$r[A \rightarrow_{[u,k]} B]v \;=\; \begin{cases} \{\} & \alpha_k(v'') \in \mathcal{V}(\mathcal{R}), \\ u \cdot w \cdot (v - v'') \;\mid & \alpha_k(v'') = \beta_k(w), \quad if \;\; v > u \\ & v' \leq v, v' = u \cdot v'' \\ \emptyset & if \;\; u = v \\ v & otherwise \end{cases}$$

Definition 3.8 *The residue map for narrowing with respect to the rewriting system $\mathcal{R}$ is $n[A \rightsquigarrow_{[u,k,\delta]} B]v = r[\delta A \rightarrow_{[u,k]} B]v$ for all $v \in \bar{\mathcal{O}}(A)$*

The definition of residue map seems much more technical than the underlying intuition. When a term A is rewritten at occurrence u, all outer subterms are unaffected and their occurrences remain unchange. Subterm at any inner occurrence v ($u < v$) still 'resides' in the resulting term B iff A/v itself is 'untouched' during the rearrangement process. The most common way to achieve this, without loss of generality, is unifying either A/v or its superterm with a variable in the left hand side of a rule. Residue of such subterm can easily be located graphically by tracing the arrows in Fig 3.2. As a remark, if the inner part of the left hand side of some rule $\alpha_k \rightarrow \beta_k$ contains a defined function 'f', any term that is narrowable using the k^{th} rule may also be narrowable at an inner occurrence using the rules that define 'f'. In such a case, a term is narrowable at two different but dependent occurrences u and v (say $v < u$) Non-overlapping property ensures that

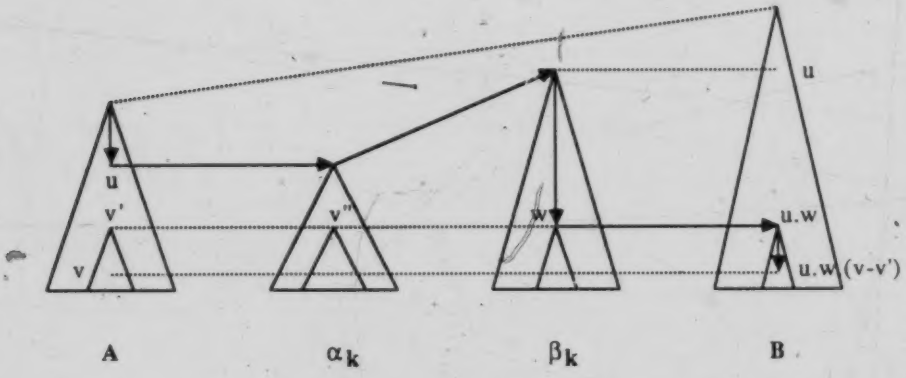


Figure 3.2: Residue Map

each narrowing derivation generates uncomparable solutions (see [You89] for details). Omitting any one of them may result in loss of a legitimate solution. However, constructor-based term rewriting systems eliminate such possibility so that the completeness and minimality results are still preserved after the rearrangement process.

Lemma 3.9 *Given a constructor-based term rewriting system, suppose a term A is narrowable at occurrence u*

$$A \leadsto_{[u,j,\rho_1]} B \leadsto_{[v,k,\rho_2]} D_1$$

If A is also narrowable at v $(v < u)$ using the k^{th} rule such that $\alpha_k/u - v \neq A/u$, then there exists a narrowing derivation

$$A \leadsto_{[v,k,\delta_1]} C \leadsto_{[U,J,\delta_2]} D_2$$

where $U = n[A \leadsto_{[v,k,\delta_1]} C]u$, J be the rule indexes of all j such that $\delta_2\delta_1 \leq \rho_2\rho_1$, and there exists a substitution τ with $\tau(D_2) = D_1$.

The lemma is in principal a formalization of Outer-Before-Inner property. Consider the case when u <> v, the residue map for narrowing U = n[$A \leadsto_{[v,k,\delta_1]} C$]u is the singleton {u}. Unlike Lemma 3.9, only one redex can be selected from this set. i.e.

$$A \leadsto_{[u,j,\rho_1]} B \leadsto_{[v,k,\rho_2]} D_1$$
$$A \leadsto_{[v,k,\delta_1]} C \leadsto_{[u,j,\delta_2]} D_2$$

Since the only possible link between two independent subterms is shared variable, the essential difference of these two derivations is the order in which variable substitutions are introduced. It simply implies that the composite substitutions must be identical (i.e. $\delta_2\delta_1 = \rho_2\rho_1$). Thus , the resulting terms must be equaled (i.e. $D_1 = D_2$). This property is similar to the independence of computation rule of Prolog. The branching factor for a set of independent redexes is one, making it possible to fix a selection rule for outer-narrowing.

Lemma 3.10 *Suppose* $A \leadsto_{[u,k,\rho_1]} B \leadsto_{[v,j,\rho_2]} D$ *where* $v <> u$ *and* v *is lexicographically less than* u, *then there exists a narrowing derivation* $A \leadsto_{[v,j,\delta_1]} C \leadsto_{[u,k,\delta_2]} D$ *such that* $\delta_2\delta_1 = \rho_2\rho_1$.

To transform any narrowing derivation to its corresponding outer-narrowing derivation, Lemma 3.9 and Lemma 3.10 are repeatedly applied . Since narrowing is complete for E_c-equality, completeness of outer-narrowing under E_c-equality is trivial if the normal form of each term is then constrainted to a constructor term.

Theorem 3.11 *Given two terms* A *and* B, *for any narrowing derivation*

$$H(A,B) = A_0 \leadsto_{[u_0,k_0,\rho_0]} A_1 \leadsto \cdots \leadsto_{[u_{n-1},k_{n-1},\rho_{n-1}]} A_n = H(C_1,C_2)$$

such that C_1 *and* C_2 *are constructor terms unifiable by a most general unifier* τ, *there exists an outer-narrowing derivation*

$$H(A,B) = A_0' \leadsto_{[v_0,j_0,\delta_0]} A_1' \leadsto \cdots \leadsto_{[v_{m-1},j_{m-1},\delta_{m-1}]} A_m' = H(D_1,D_2)$$

such that D_1 *and* D_2 *are constructor terms unifiable by a most general unifier* θ *so that* $\theta\delta_{m-1}\ldots\delta_0 \leq \tau\rho_{n-1}\cdots\rho_0$.

Non-overlapping property of constructor-based rewriting systems plays a key role in establishing the minimality result of outer-narrowing. Theorem 3.11

shows that complete set of E_c-unifiers is effectively enumerated through outer-narrowing, and §3.3 suggests that the substitutes of each unifier are normalized. Again, two unifiers are comparable by $\leq_E$ iff they are comparable by $\leq$. Given two independent outer-narrowing derivations, branching will eventually occur leading to uncomparable solutions. That is exactly where non-overlapping property enters into the picture. Assume the two outer-narrowing derivations branch at the i^{th} step (i.e. $u_h = v_h$, $k_h = j_h$, $\rho_h = \delta_h$ and $A_h = B_h$ for $0 \leq h \leq i$):

$$H(A,B) = A_0 \leadsto_{[u_0,k_0,\rho_0]} A_1 \leadsto \ldots \leadsto_{[u_{n-1},k_{n-1},\rho_{n-1}]} A_n = H(C_1,C_2)$$

$$H(A,B) = B_0 \leadsto_{[v_0,j_0,\delta_0]} B_1 \leadsto \ldots \leadsto_{[v_{m-1},j_{m-1},\delta_{m-1}]} B_m = H(D_1,D_2)$$

1. Along with left-linearity, which guarantees that all substitutes of an E_c-unifier are extracted from the left hand side of some rule, uncomparable solutions are generated if the term A_i is rewritten at the same occurrence ($u_i = v_i$) but with different rules (i.e. $A_i/u_i = B_i/v_i$ but $k_i \neq j_i$).

2. Consider the case if A_i is narrowed at two different but dependent occurrences (say u_i and v_i with $u_i < v_i$), and the k^{th} rule is used to narrow the outer occurrence u_i so that the function symbol contained in α_{k_i} coincides with that of B_i/v_i (i.e. $\alpha_{k_i}/(v_i - u_i) = B_i/v_i$). Since

B_i/v_i is also reducible, so $B_i(v_i) \in \mathcal{F}_d$. Hence, $\alpha_{k_i}/(v_i - u_i) \in \mathcal{F}_d$, which contradicts with the constructor-based restriction.

3. Consider the general case for which A_i is narrowed at two different but dependent occurrences (with $u_i < v_i$). As mentioned before, the narrowing step at an inner occurrence v_i must in some way contribute to the reducibility of the outer occurrence u_i. Since A_i/u_i is a function term (say $f(t_1, \ldots, t_n)$) and C_1, C_2, D_1, D_2 are constructor terms, the function symbol 'f' must be narrowed at a later narrowing step. Assume

$$A_i \leadsto_{[u_i,k_0,\rho_i]} A_{i+1} \leadsto \ldots \leadsto_{[u_{n-1},k_{n-1},\rho_{n-1}]} A_n$$

$$B_i \leadsto_{[v_i,j_i,\delta_i]} \overset{*}{\leadsto} B_l \leadsto_{[v_l,j_l,\delta_l]} B_{l+1} \leadsto \ldots \leadsto_{[v_{m-1},j_{m-1},\delta_{m-1}]} B_m$$

so that $v_l = u_i$ and $B_l(v_l) = \text{'f'} = A_i(u_i)$. By outer-before-inner property, no defined function symbol may appear in α_k. The only possible way to narrow A_i (or B_i) at two dependent occurrences ($u_i < v_i$) is that B_i/v_i or its superterm unifies with a variable in α_k. By Lemma 3.9, outer-before-inner rearrangement is possible for the narrowing derivation $H(A, B) = B_0 \overset{*}{\leadsto} B_m = H(D_1, D_2)$, i.e. contradicts with the initial assumption that it is an outer-narrowing derivation.

Theorem 3.12 *Given two unrelated outer-narrowing derivations of two unifiable terms A and B,*

$$H(A,B) = A_0 \leadsto_{[u_0,k_0,\rho_0]} A_1 \leadsto \ldots \leadsto_{[u_{n-1},k_{n-1},\rho_{n-1}]} A_n = H(C_1,C_2)$$

$$H(A,B) = A_0' \leadsto_{[v_0,j_0,\delta_0]} A_1' \leadsto \ldots \leadsto_{[v_{m-1},j_{m-1},\delta_{m-1}]} A_m' = H(D_1,D_2)$$

where C_1 and C_2 are constructor terms unifiable by a most general unifier τ, and D_1 and D_2 are constructor terms unifiable by a most general unifier θ so that $\tau\rho_{n-1}\ldots\rho_1$ and $\theta\delta_{m-1}\ldots\delta_1$ do not compare by $\leq_E$.

Since outer-narrowing subsumes all other narrowing derivations and the solutions obtained as such are uncomparable, complete set of minimal E_c-unifiers is effectively enumerated through outer-narrowing.

Chapter 4

Conditional Rewriting System

In this chapter, we propose a conditional term rewriting system that can be implemented efficiently. This system is heavily inspired by the work of [You88]. Confining to E_c-equality, the completeness and minimality characteristics of the proposed system are investigated. We further define an extended unification scheme that can be easily incorporated into existing Prolog systems.

4.1 Syntax

Syntax of the proposed conditional rewriting system is similar to that of Horn clause logic programs. The building blocks are constants, variables, predicates and functors. The functors $\mathcal{F}$ are further divided into disjointed

sets of constructors $\mathcal{F}_c$ and defined function symbols $\mathcal{F}_d$. Notions of atoms, terms and constructor terms are carried over from preceding chapters. A basic rewrite rule is of the form

$$l \rightarrow r :- P_1, \ldots, P_n$$

where

l and r are terms, $P_1, \ldots, P_n$ ($n \geq 0$) are predicates.

$P_1, \ldots, P_n$ are collectively referred to as the condition for which l is rewritten to r. A rule without any condition is an unconditional rewrite rule. A legitimate conditional term rewriting system $\mathcal{C}$ is a set of such rewrite rules which complies with the following two constraints:

1. The unconditional portion of each rewrite rule constitutes a constructor-based term rewriting system.

2. For each rule in $\mathcal{C}$, no extra variable is introduced in the condition $P_1, \ldots, P_n$.

An n-ary predicate is treated as a boolean function which maps its arguments to a set of special constructors {**true**, **false**}. The condition $P_1, \ldots, P_n$, representing the predicate $\mathbf{and}(P_1, \mathbf{and}(\ldots, \mathbf{and}(P_{n-1}, P_n)\ldots))$, denotes a form of cascading the special boolean function **and** which is then interpreted as

$$\text{and(X, } \mathbf{false}) \rightarrow \mathbf{false}$$

$$\text{and(}\mathbf{false}\text{, X)} \rightarrow \mathbf{false}$$

$$\text{and(}\mathbf{true}\text{, }\mathbf{true}\text{)} \rightarrow \mathbf{true}$$

Hence, a rewrite rule $l \rightarrow r :- P_1, \ldots, P_n$ is interpreted as $l \rightarrow r :- P_1 \stackrel{*}{\rightarrow} \mathbf{true}, \ldots, P_n \stackrel{*}{\rightarrow} \mathbf{true}$. Intuitively, the left hand side l is rewritten to its corresponding right hand side r if all the predicates $P_1, \ldots, P_n$ are rewritten to **true**. A fact P is just the syntactic sugar of an unconditional rewrite rule $P \rightarrow \mathbf{true}$, and a Horn clause of the form $H :- B_1, \ldots, B_n$ is interpreted as $H \rightarrow \mathbf{true} :- B_1 \stackrel{*}{\rightarrow} \mathbf{true}, \ldots, B_n \stackrel{*}{\rightarrow} \mathbf{true}$. This configuration permits co-existence of both conditional rewrite rules and Horn clauses, and amalgamates them within a purely logical framework.

4.2 Completeness

The system described in §4.1 is an instance of type III_n conditional rewriting system [BK86]. It is proved confluent without resorting to the termination property. Since the system may not even terminate at all, all those elegant completeness results for canonical term rewriting systems [Fay79, Hul80, Kap84] may not be applicable in this context. Without loss

of generality, our discussion is concentrated on rewrite rules with at most one predicate appearing in the condition.

Definition 4.1 *A term A is (conditionally) reduced to B, written $A \rightarrow_C B$, iff there exists a rule $l \rightarrow r :- P$ in C, a non-variable occurrence u of A and a substitution δ such that* $A/u = \delta l$, $\delta P \stackrel{*}{\rightarrow}_C \mathbf{true}$ *and* $B = A[u \leftarrow \delta r]$.

The conditional rewriting relation $\rightarrow_C$ associated with C is the finest relation over $T(\mathcal{F}, \mathcal{V})$ containing C and closed under substitution and replacement. The reflexive and transitive closure of $\rightarrow_C$ is denoted by $\stackrel{*}{\rightarrow}_C$. It is well established that a term rewriting system possesses the Church-Rosser property iff it is confluent. Since classic E-equality "$=_E$" is equivalent to the symmetric closure of rewriting relation "$\stackrel{*}{\rightarrow}$" [HO80], equality between two different terms can be verified through Church-Rosser property of the underlying system. In particular, $A =_E B$ iff there exists a term C such that $A \stackrel{*}{\rightarrow} C$ and $B \stackrel{*}{\rightarrow} C$. Reduction, which amounts to one-sided simplification, cannot generate variable bindings for a given goal statement. Instead, narrowing is employed to obtain the substitution that satisfies the given goal.

Definition 4.2 *A term A is (conditionally) narrowed to B, written* $A \leadsto_\delta B$, *iff there exists a rule* $l \rightarrow r :- P$ *in* $\mathcal{C}$, *a non-variable occurrence u of A and a most general unifier* δ' *for* A/u *and l (i.e.* $\delta'(A/u) = \delta' l$*). such that* $\delta' P \leadsto_{\delta''}$ **true**, $\delta = \delta'' \cdot \delta'$ *and* $B = \delta(A[u \leftarrow r])$.

As mentioned before, all those elegant completeness results may not hold for non-terminating rewriting systems. The major hurdle is the existence of non-normalizable solutions in these systems. An implication of Hullot's result [Hul80] is that in absence of termination, narrowing is still complete with respect to normalized solutions for confluent term rewriting systems. [GM86] further suggests that under a stronger notion of level-confluence, conditional narrowing is complete with respect to normalized solutions that are not "dependent" on some other non-normalizable solutions (even in the presence of extra variables). Stealing an example from [GM86], consider the following level-confluent term rewriting system in the presence of extra variables,

Eg. 4.1.,

Given a T.R.S.,

$$h \rightarrow d(h)$$
$$f(a) \rightarrow b :- X \rightarrow d(X)$$

Naive conditional narrowing is not even complete with respect to normalized solution, although $\mathcal{C} \models_E \delta(f(Z)) = b$ with $\delta = \{Z/a\}$. This requires narrowing to compute the non-normalized solution $\gamma = \{X/h\}$ for the introduced variable, which is generally impossible under E-equality.

Observe that the substitute for any extra variable is only used during proof construction but never contributed to the normalized **output** solution $\{Z/a\}$. This kind of dependency is sufficiently pruned if no extra variable is allowed, so that any non-variable substitute must be extracted from the left hand side of a rule. In such a case, naive conditional narrowing is complete with respect to normalized solution [LPB+87].

Eg. 4.2.,

Given a T.R.S.,

$$h \rightarrow d(h)$$
$$f(X) \rightarrow b :- X \rightarrow d(X)$$

Although $\mathcal{C} \models_E \delta(f(Z)) = b$, conditional narrowing cannot yield the non-normalizable solution $\delta = \{Z/h\}$.

In summary, in absence of termination, conditional narrowing is in general incomplete for confluent rewriting systems confined to E-equality.

4.2.1 E-equality vs E_c-equality

In this section, we focus our investigation on a subset of E-equality, namely E_c-equality, in quest of the completeness results for conditional narrowing. Recall that $A =_{E_c} B$ iff there exists a constructor term C such that $A =_E C$ and $B =_E C$. From now on, we refer to a "system" as a conditional term rewriting system described in §4.1. To illustrate the difference between E-equality and E_c-equality, consider the following two systems:

Eg. 4.3.,

Given a T.R.S.,

$$h \rightarrow d(h)$$
$$d(X) \rightarrow a$$
$$f(X) \rightarrow g(X)$$

Given a goal $?\delta(f(X)) = \delta(g(d(X)))$

1. Under E-equality, $\mathcal{C} \models_E \delta(f(X)) = \delta(g(d(X)))$ with $\delta = \{X/h\}$ or $\delta = \{X/a\}$.

2. Under E_c-equality $\mathcal{C} \models_{E_c} \delta(f(X)) = \delta(g(d(X)))$ with $\delta = \{X/h\}$ or $\delta = \{X/a\}$.

In both cases, although conditional rewriting cannot produce the non-normalizable solution (i.e. $\{X/h\}$), it is still complete in the sense that $h =_E a$.

Eg. 4.4.,

Given a T.R.S.,

$h \rightarrow d(h)$

$f(X) \rightarrow g(X)$

Given a goal $?\delta(f(X)) = \delta(g(d(X)))$

1. Under E-equality, $\mathcal{C} \models_E \delta(f(X)) = \delta(g(d(X)))$ with $\delta = \{X/h\}$ since $f(h) \rightarrow g(h) \rightarrow g(d(h))$ (i.e. $f(h) =_E g(d(h))$). Again conditional narrowing cannot generate this solution.

2. Under E_c-equality, although $f(h) \rightarrow g(h) \rightarrow g(d(h))$, the substitution $\delta = \{X/h\}$ does not constitute a solution as $g(d(h))$ isn't a constructor term. In fact, for any non-normalizable substitute t, it can be a solution only if a

rule of the form $t \rightarrow d(t)$ is in $\mathcal{C}$. Further narrowing of t results in a non-terminating derivation which implies that t is not E_c-equivalent to any constructor term (thus even itself). Hence, $\mathcal{C} \not\models_{E_c} \delta(f(X)) = \delta(g(d(X)))$

In this case, conditional narrowing is complete under E_c-equality but not under E-equality.

Completeness of narrowing is best manifested by its ability to enumerate all the correct answers of a given goal. Tackling this issue from the opposite direction, we analyze characteristics of the solution set under E_c-equality. By definition, a substitution δ is an E_c-unifier of two terms A and B if both δA and δB are rewritten to an identical constructor term. Theoretically, the solution need not be a constructor term, and it can even be a non-normalizable term. In a slightly more general setting, [You88] implies that if a solution δ_1 is non-normalizable, then another solution δ_2 which is normalized **must** also exist such that $\delta_2 \leq_E \delta_1$. It accounts for the fact that the solution set can never consist of non-normalizable terms **only**. The solution set can now be viewed as a partition of distinguished equivalent classes. Two solutions are in the same class iff they are comparable by $\leq_E$. Within each class, there exists a normalized solution which is more general than all the others. It doesn't matter if a non-normalizable solution exists or

not. What really matters is the co-existence of a most general normalized solution which can be enumerated through narrowing. Conditional rewriting system is essentially an unconditional rewriting system augmented with condition solving. A candidate rule (i.e. a rewrite rule whose left hand side is unifiable with a function term) is applied only if its associated condition $\mathcal{P}$ is resolved. Since naive conditional narrowing is complete for confluent rewriting systems (with respect to normalized solutions) [LPB+87] and outer-narrowing subsumes naive narrowing, we claim that conditional narrowing under E_c-equality is complete for the conditional rewriting systems described in §4.1.

Claim 4.3 *For any set of E_c-unifiers τ of two terms, A and B, there exists a conditional narrowing derivation*

$$H(A,B) \overset{\cdot}{\sim}_{c\ [U,K,\theta]} H(C_1,C_2)$$

where C_1 and C_2 are constructor terms unifiable by a most general unifier γ such that $\gamma \cdot \theta$ is an E_c-unifier for A and B with $\gamma \cdot \theta \leq_E \tau \uparrow \mathcal{V}(A,B)$.

Our claim is justified by the fact that any such system $\mathcal{C}$ can be transformed to an equivalent unconditional rewriting system $\mathcal{R}$. The systems are equivalent in the sense that any two terms are unifiable in $\mathcal{C}$ iff they are unifiable in $\mathcal{R}$, and the solution set can be enumerated effectively in both systems. The

principal transformation technique is derived from [DP88, GM87]. For any conditional rewrite rule

$$l(\bar{x}) \rightarrow r(\bar{x}) :- P(\bar{x})$$

where

$\bar{x}$ is the set of variables appear in l which allows subterm distribution from l to r.

A new function symbol $P' \notin \mathcal{F}$ is introduced. The above rule is then replaced by two unconditional ones:

$$l(\bar{x}) \rightarrow P'(\bar{x}, P(\bar{x}))$$
$$P'(\bar{x}, \mathbf{true}) \rightarrow r(\bar{x})$$

Recall that the unconditional portion of $\mathcal{C}$ is constructor-based and each P' is a new function symbol. After all the rules are transformed, the resulting system must also be an unconditional constructor-based system.

Eg. 4.5.,

Given a system $\mathcal{C}$,

$$l_1(\bar{x}) \rightarrow r_1(\bar{x}) :- P_1(\bar{x})$$
$$l_2(\bar{x}) \rightarrow r_2(\bar{x}) :- P_2(\bar{x})$$
$$l_3(\bar{x}) \rightarrow r_3(\bar{x})$$

it is transformed to an unconditional system $\mathcal{R}$,

$$l_1(\bar{x}) \rightarrow {P_1}'(\bar{x}, P_1(\bar{x}))$$
$${P_1}'(\bar{x}, \mathbf{true}) \rightarrow r_1(\bar{x})$$
$$l_2(\bar{x}) \rightarrow {P_2}'(\bar{x}, P_2(\bar{x}))$$
$${P_2}'(\bar{x}, \mathbf{true}) \rightarrow r_2(\bar{x})$$
$$l_3(\bar{x}) \rightarrow r_3(\bar{x})$$

Consider the above example, if $P_1(\bar{x})$ cannot be narrowed to **true**, $l_1(\bar{x})$ is 'irreducible' in $\mathcal{C}$. At a first glance, the unconditional counterpart of $\mathcal{C}$ seems to have a different outcome: $l_1(\bar{x})$ is now rewritten to ${P_1}'(\bar{x}, P_1(\bar{x}))$. Observe that ${P_1}'$ is a defined symbol, making ${P_1}'(\bar{x}, P_1(\bar{x}))$ a function term rather than a constructor term. The semantics of E_c-equality guarantees that $l_1(\bar{x})$ is also 'irreducible' in $\mathcal{R}$. If $P_1(\bar{x})$ is solvable, then ${P_1}'(\bar{x}, P_1(\bar{x}))$ is rewritten to ${P_1}'(\bar{x}, \mathbf{true})$. Hence, $l_1(\bar{x})$ is narrowable to $r_1(\bar{x})$ in both $\mathcal{C}$ and $\mathcal{R}$. As ${P_1}'$ is a distinct function symbol defined by exactly one rule, the solution set for $\bar{x}$ must be **identical** in both systems. Analogue to §3.3, the non-variable substitutes of any E_c-unifier so obtained thru conditional narrowing are normalized, leading to the following claim.

Claim 4.4 *Complete and minimal sets of E_c-unifiers always exists for any two E_c-unifiable terms.*

4.3 Extended Unification

The heart of traditional logic programming is resolution-based syntactic unification. In contrast to conventional procedural programming paradigm, pattern matching with no specific input/output mode enhances functional reversibility. The syntactic nature, on the other hand, limits the scope of logic programming to first-order languages. The key to any successful functional logic language is an extended version of naive unification scheme which allows function evaluation within first-order 'predicates'. Fig 4.1 presents a simplified syntactic unification algorithm without occurs check. Two terms, A and B, are dereferenced to their corresponding 'values' before unification. If either one of them is a free-variable, it is then bound to the remaining term. If both A and B are function terms with the same principal functor and same arity, then their corresponding argument pairs are successively unified; otherwise, the inference system halts and returns with failure. Execution of a logic program is initiated by a given goal. The computation mimics 'a proof construction which generates the required substitution as a side effect'. A version of such solver algorithm is depicted

```
syntactic_unify(A, B)
  {
    dereference(A)
    dereference(B)

    if ( A is a variable )
      {
        θ = {A/B} ∪ θ
        return(success)
      }
    if ( B is a variable )
      {
        θ = {B/A} ∪ θ
        return(success)
      }
    if ( A = f(A_1, ..., A_n) and B = f(B_1, ..., B_n) )
      {
        for i = 1 to n
          {
            if not( syntactic_unify(A_i, B_i) )
              return(failure)
          }
        return(success)
      }

    return(failure)
  }
```

Figure 4.1: Syntactic unification without occur check

in Fig 4.2. It is well known as 'ABC algorithm' [vE82] since the algorithm is divided into (A) subgoal selection, (B) unification and (C) backtracking handling. The function select() returns a subgoal from a set of unsolved predicates as the currently activated call. The set of candidate clauses, whose heads are potentially unifiable with current call, is generated by clauseGenerator(). Successive clauses of this collection are enumerated by the function nextClause() which are then unified with the activated call. The resolvent is obtained if unification succeeds; otherwise, the function reset() restores all mandatory information required to explore alternative searching paths.

In §4.2.1 we shown that narrowing is complete for conditional rewriting systems under E_c-equality. Its operational semantics follows the trails of outer-narrowing which subsumes all other narrowing strategies. The inference mechanism must abide with two essential properties: (i) leftmost subgoal selection and (ii) outer-before-inner term rewriting. Although unsolved subgoals are treated as an unordered set in general unification scheme, leftmost selection rule is generally employed in most Prolog systems. An implication of outer-before-inner rewriting is that any narrowable outer-occurrence of a function term **must** be rewritten before its inner occurrence is narrowed. This property can easily be incorporated into existing syntactic unification by means of outermost-to-innermost scanning. For any two function terms A and B, function symbol matching starts from

```
naive_solve(G)
  {
    A: if ( G = nil )
         return(success)
       else
         {
           curCall ← select(G)
           curGen ← clauseGenerator(curCall)
         }
    B: curClause ← nextClause(curGen)
       if ( curClause ≠ nil )
         {
           if ( syntactic_unify(curCall, head(curClause)) )
             {
               G ← resolvent(G, curClause, θ)
               goto A
             }
           else goto B
         }
    C: if ( curBak = nil )
         return(failure)
       else
         {
           reset(curBack)
           goto B
         }
  }
```

Figure 4.2: Naive ABC algorithm

outermost functor to innermost functor. Unification succeeds if A and B are syntactically unifiable; otherwise, they must disagree at some occurrence u. In this case, the process is suspended and replaced by outer-narrowing of A/u and B/u. If they can be narrowed to two syntactical unifiable constructor terms, then A and B are unifiable and unification resumes. Extended unification fails if any one of A or B cannot be rewritten to a constructor term, or even, their corresponding 'fully narrowed' terms are non-unifiable. Obviously, syntactic unification must be distinguished from extended unification, and outer-narrowing within extended unification is like activating a mini-Prolog inside a proof construction. The modified unification scheme and solver algorithm are presented in Fig 4.3 and Fig 4.4 respectively.

Eg. 4.6.,

1. unify(f(Y), f(g(X)) always succeeds.

2. unify(f(g(X), f(h(X)) leads to functor conflict at u = 1 (since the outer function symbol 'f' is identical). it is then suspended and replaced by successive outer-narrowing of the subterms g(X) and h(X). If g(X) and h(X) are narrowed to syntactical unifiable constructor terms, then unify(f(g(X), f(h(X))) resumes and succeeds.

```
unify(A, B, syntactic_unify)
  {
    dereference(A); dereference(B)

    if ( A is a variable )
      {
        θ = {A/B} ∪ θ
        return(success)
      }
    if ( B is a variable )
      {
        θ = {B/A} ∪ θ
        return(success)
      }
    if ( A = f(A_1, ..., A_n) and B = f(B_1, ..., B_n) )
      {
        for i = 1 to n
          if not( unify(A_i, B_i, syntactic_unify) ) goto NEXT
        return(success)
      }
    NEXT:
    if ( syntactic_unify ) return(failure)
    else
      {
        [narrowed, A'] ← solve(A)
        if not(narrowed) return(failure)
        [narrowed, B'] ← solve(B)
        if not(narrowed) return(failure)
        return(unify(A', B', true))
      }
  }
```

Figure 4.3: Extended unification scheme

```
solve(G)
  {
    A: if ( G = nil )
         return(success, G)
       else
         {
           curCall ← select(G)
           curGen ← clauseGenerator(curCall)
           if ( curGen = nil )
             {
               if ( curCall is a rewritten term )
                 return(success, curCall)
             }
         }
    B: curClause ← nextClause(curGen)
       if ( curClause ≠ nil )
         {
           if ( unify(curCall, leftHandSide(curClause), false) )
             {
               G ← resolvent(G, curClause, θ)
               goto A
             }
           else goto B
         }
    C: if ( curBak = nil )
         return(failure)
       else
         {
           reset(curBack)
           goto B
         }
  }
```

Figure 4.4: Solver algorithm

Recall that the system is complied to the restrictions imposed in §4.1 so that the unconditional portion of $\mathcal{C}$ constitutes a constructor based term rewriting system. Although this modified unification scheme can be used in a slightly more general setting of equation solving, incorporating it into a practical programming language is relied on the constraints of $\mathcal{C}$. Remember that a fact P is just the syntactic sugar of $P \rightarrow \mathbf{true}$ and is treated like any other function terms. It has two affects on the extended unification scheme:

1. Given a rule $l \rightarrow r :- P$, the condition P is proved before l is rewritten to r. If a subgoal $?G$ is unifiable with l, the coherent treatment of rewrite rules and predicates allows us to replace $?G$ by $?P,r$.

2. Since computation is initiated by a goal statement, one of the arguments of unify(A,B), at any time, must be extracted from the left hand side of some rule. Assume B is always selected as such, non-overlapping property of $\mathcal{C}$ guarantees that B must not unifiable with the left hand side of any rule. In case of functor conflict, unify(A,B) needs to outer-narrow A only (instead of both A and B).

The simplied proof procedure incorporating these two effects is as follows:

1. Select the leftmost subgoal G_1 from the goal $G \leftarrow G_1, \ldots, G_m$ as the next subgoal to solve.

2. Arbitrarily select a rewrite rule $l \rightarrow r :- P_1, \ldots, P_n$ from $\mathcal{C}$ whose left hand side l is potentially unifiable with G_1.

3. Obtain the most general unifier δ of G_1 and l when

   (a) l and G_1 are syntactically unifiable.

   (b) l and G_1 disagree on function symbol at occurrence u

      (i) Suspend the unification of l and G_1.

      (ii) Outer-narrow G_1/u to a constructor term and syntactically unify it with l/u.

4. Replace G_1/u in G by $P_1, \ldots, P_n, r$. Apply the unifier δ to it to obtain a new goal $\delta(P_1, \ldots, P_n, G_1[u \leftarrow r], G_2, \ldots, G_m)$.

5. Repeat steps 1-4 until either

   (a) The goal is a null clause or

   (b) No clause can be selected in step 2 or

   (c) In step 3b, G_1 cannot be narrowed to a constructor term which is syntactically unifiable with l/u.

Before presenting an example to illustrate the proof construction procedure, the notations used in the following example are explained. If there is a functor

conflict, $[A;B]$ is used to denote the process of outer-narrowing A and then syntactically unifying the 'fully narrowed term' with B. The predicate in bold face is the currently activated call of a goal statement.

Eg. 4.7.,

Given a system $\mathcal{C}$,

a(X) → b(X) : − c(X)

b(X) → d(X) : − e(X)

f(d(X))

c(X)

e(X)

? **f(a(X))**

? [**a(X)**; d(X)]

? [**c(X)**,b(X); d(X)]

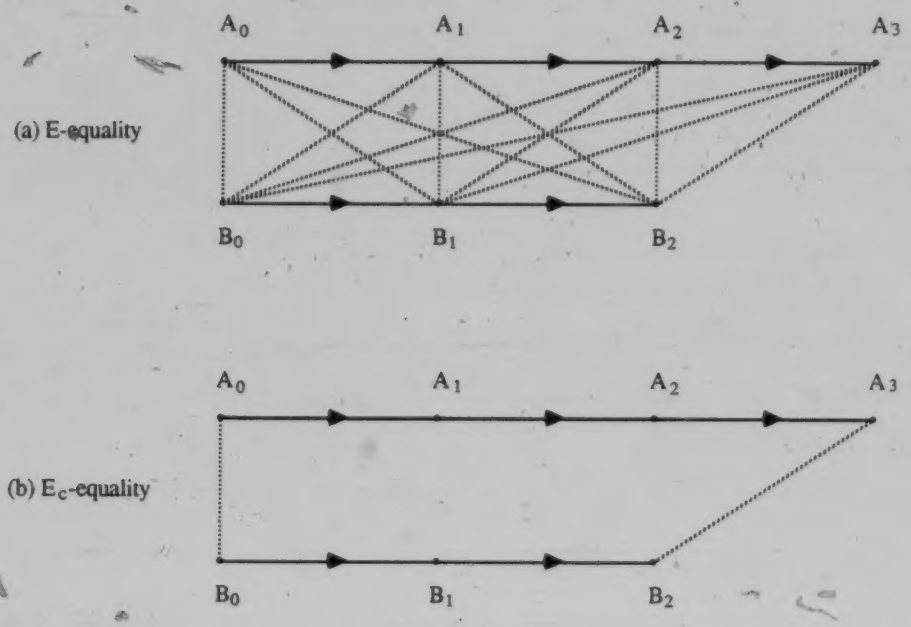
? [**b(X)**; d(X)]

? [**e(X)**,d(X); d(X)]

? [**d(X)**; d(X)]

? {}

An important contribution of this extended unification scheme under E_c-equality is the existence of efficient and feasible implementation. In this setting, equality of two terms is proved by outer-narrowing them all the way to two constructor terms before syntactic unification is attempted (if the constructor terms really exist). A major impediment of E-equality is the cross-unification of intermediate results. Any underlying implementation must account for this aspect and 'store up' all intermediate results for future consideration. The improvement is best illustrated in Fig 4.5. Under E-equality, proving $A_0 =_E B_0$ leads to subsequent cross-unification of A_i and B_j. All the intermediate results, (A_0, A_1, A_2, A_3) and (B_0, B_1, B_2), must be stored for future consideration of $A_i =_E B_j$. On the other hand, E_c-equality required syntactic unification of A_3 and B_2 in order to establish $A_0 =_{E_c} B_0$.



Each arrow represents a narrowing step, and the dotted line denotes possibly cross-unification of two (intermediate) terms

Figure 4.5: Improvement over cross-unification

Chapter 5

Functional Logic Programming

This chapter describes various implementation issues of the extended unification scheme, in particular, porting such scheme to existing Prolog systems. We choose Waterloo Unix Prolog (WUP) as the target language since it adequately demonstrates general concepts of memory organization, data structures and control mechanism used in most systems. The first part is an introductory overview of WUP to help understand fundamental techniques regarding Prolog implementation. Readers interest in more in-depth analysis should refer to [Hog84, Cam84]. The second part is devoted to incorporating extended unification into WUP resulting in a modified language named FLP (an acronym for Functional Logic Programming).

5.1 Overview of WUP

Waterloo Unix Prolog (WUP) is a Prolog dialect written in C under UNIX[1] operating system [Che84]. Its most noticeable features are separate compilation, adaptable runtime support, Prolog clauses database management and modular programming. This integrated environment allows the users to edit, execute, develop and debug large Prolog programs. Adaptable runtime support, specifically tailored to certain programming environments, together with separate compilation enhance incremental program development without sacrificing portability of the system. Prolog clauses management coupled with modular programming facilitates sophisticated yet flexible means to maintain various components of large Prolog programs. Each program is partitioned into modules or disjointed set of clauses. These modules reside in different files, directories and even file systems. Applying the same concept in Modula-2 [Wir82], users can implicitly or explicitly export a module making it 'accessible' or 'visible' to the other modules. The entire collection of modules are organized into a tree-like hierarchical structure. WUP also defines and controls several module searching strategies to enumerate the set of candidate clauses whose heads are unifiable with a given predicate.

[1]UNIX is a Trademark of Bell Laboratories

5.1.1 Memory Management

Conventional implementations of programming languages divide program memory into two different segments: data segment and control segment. The data segment or heap is created before program execution. It contains the codes of a program, usually in a compiled or compacted byte-code format, which are accessible by the compiler. Control segment, which expands or retracts dynamically, stores necessary information to locate the computation path and to identify variable bindings along the path.

The runtime structure of WUP is made up of three stacks: runtime stack, copy stack and trail stack. Runtime stack performs same tasks as control segment. A stack node is created whenever the activated call successfully unifies with the head of any candidate clause. Two types of object, stack frame and environment, which constitute a stack node are pushed on top of the stack. The stack frame maintains navigation information of the matched clause-head pair. Its size depends on whether it is a deterministic node or non-deterministic node (see §5.1.3). The environment, with size equals to the number of unique variables in the matched clause, records the binding of each variable. The copy stack is essentially a heap containing codified version of all the clauses. Clauses defining the same predicate (i.e. same name and arity) are compiled and linked together according to their textual appearances in the program. However, the copy stack is not exactly a

heap since it dynamically saves newly created terms at runtime (see §5.1.2). Trial stack records pointers to the newly bound variables which must then reset upon backtracking. By the nature that all deterministic stack nodes (including the associated environment) beyond the most recent backtracking point are successively poped during the course of backtracking, only those variable bindings below the most recent non-deterministic backtracking node are recorded.

5.1.2 Data Structure

The primitive construct of WUP internals or kernel is PC_WORD. It is a structured record with two fields, a tag and a value. The tag field identifies what kind of object a PC_WORD represents. Depending on the type and the context a PC_WORD is being used, the value field stores either an integer, a floating point number, a character constant or a pointer to another PC_WORD. The declaration and molecular structure of a PC_WORD is illustrated in Fig 5.1. Control information retained in the runtime stack are defined as collections of PC_WORD. Prolog clauses are preprocessed (parsed and compiled) into corresponding clusters of PC_WORD before they are linked up and placed onto copy stack. Fig 5.2 shows the PC_WORD representation of a variable functor clause head. Structured objects (functors or lists) are also represented by a continuous series of PC_WORD. A related

```
typedef struct pc_word
  {
     int tag;
     union word
        {
           int ival;
           float fval;
           struct pc_word *ptr;
           char *sval;
        }
  }
```

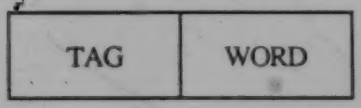


Figure 5.1: Declaration and molecular structure of PC_WORD

issue is the assignment of these structured objects to a free variable. Current Prolog implementations focus on two different approaches: structure sharing and structure copying. Structure sharing, as suggested by its name, allows the newly created objects to share a common skeleton (eg. [X | Y]). The skeleton, acting as the proxy of a structured term, is usually embedded within an input program. Program memory is substantially reduced by sharing the skeleton of program codes rather than creating new ones every time a variable is bound. Each structured object is represented by two pointers: one directs to the skeleton and the other points to its binding environment. Only two pointers are instantiated whenever a free variable unifies with a structured term. However, a structured term may contain deeply nested sub-term

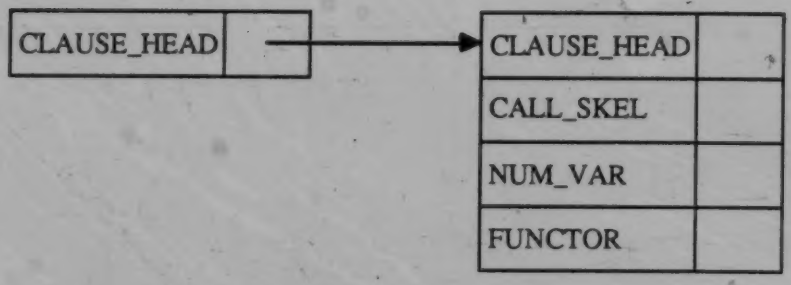


Figure 5.2: PC_WORD representation of a clause head

manifested by a long chain of PC_WORD. Despite rapid term construction, accessing such a term is inefficient as successive dereferencing is inevitable. Our target system, WUP, uses the alternative structure copying approach. During unification, a variable may bind to a free variable, a ground structure, a constant or a non-ground structure. In the first two cases, the stack node stores a single pointer to the ground structure or the variable. A constant is assigned directly to the appropriate variable environment. In the last case, a new copy of non-ground structured object is created and placed on the copy stack. A pointer to this newly constructed object is then assigned to the free variable involved in unification. Extra memory is allocated to create multiple occurrences of these objects from their original codes. Although term construction is expensive (in terms of space and time), the term is

readily accessible without a long chain of dereferencing.

Backtracking consists of cutting variable linkage created during unification and restoring necessary information to explore untried alternatives. Any reference to the retracted stack nodes results in dangling pointers. The following restrictions concerning pointer assignment are imposed to avoid dangling pointers:

1. Since the stacks are poped in a Last-In-First-Out fashion, pointers within the same stack must originate from higher order address to lower order address.

2. When a free variable unifies with a structured object, the object is copied onto the copy stack. A pointer is emanated from the variable environment to this newly created object. The copy stack must then reside below the runtime stack.

A simple memory layout of the copy and runtime stacks is depicted in Fig 5.3

5.1.3 Control Mechanism

WUP adapts typical left-to-right depth-first searching scheme which is commonly employed by most existing Prolog systems. The proof procedure is based on Van Emden's ABC algorithm [vE82] (see Fig 4.2). This algorithm assumes the data structure of a proof tree. A stack node (stack frame and

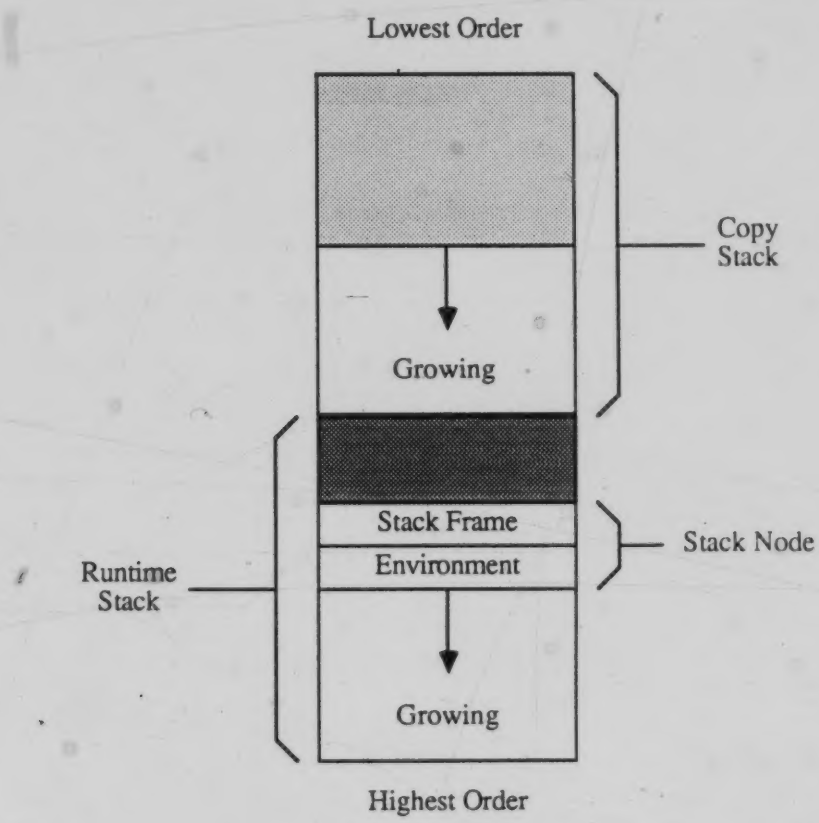


Figure 5.3: Memory layout of copy stack and runtime stack

environment) is pushed onto the runtime stack whenever unification succeeds. A sequence of stack nodes, from the root (i.e. the initial goal) up to current node, is retained in order to carry out necessary tree traversing. The minimal information stored in a stack frame consists of:

1. A pointer 'CALL' to indicate the subgoal call of the clause causing creation of the current node. Remaining unsolved subgoals (i.e. to the right of current call) within the same clause are accessible through this pointer.

2. A stack frame pointer 'FATHER' to denote the parent frame of the current node. It maintains fundamental structure of the entire proof tree. If all the subgoals within the same clause are solved (i.e. the CALL field of the current node is nil), traversing will continue with untried subgoal of its FATHER.

Non-deterministic computation occurs if current call is unifiable with more than one candidate clause. Under such circumstances, necessary information to explore untried alternative clauses is stored in the proof tree The current node is now marked as potential backtracking point. For whatever reason unification fails, all nodes from the current one down to the most recent backtracking point are poped. The stacks and environment are reseted before unification with alternative clause is attempted. A stack node must maintain additional traversing information in order to accommodate these actions:

3. A pointer 'NEXTCL' to retrieve the chain of untried alternative candidate clauses. Backtracking results in successive unification with these clauses.

4. A pointer 'BACK' to indicate previous backtracking point. A series of backtracking is made possible with the help of a global variable 'CurBack', which denotes the most recent backtracking point.

5. Two pointers 'RESET' and 'COPY' to record the top of trail stack and runtime stack respectively. During backtracking, variable bindings referenced beyond RESET are undo, and segments of copy stack are poped according to the value of COPY.

Obviously, the size of a non-deterministic stack frame is much larger than its deterministic counterpart. For the sake of memory utilization, an interpreter or compiler should have two different kinds of stack frame. The deterministic and non-deterministic stack frames of WUP are depicted in Fig 5.4.

The unification algorithm used in WUP is a table-driven routine without 'occurs check'. This table, as shown in Fig 5.5, is a two-dimensional array. Its index are various possible argument types of a goal call and the head of a clause. The entries of this table are constants which designate different actions required for successful unification.

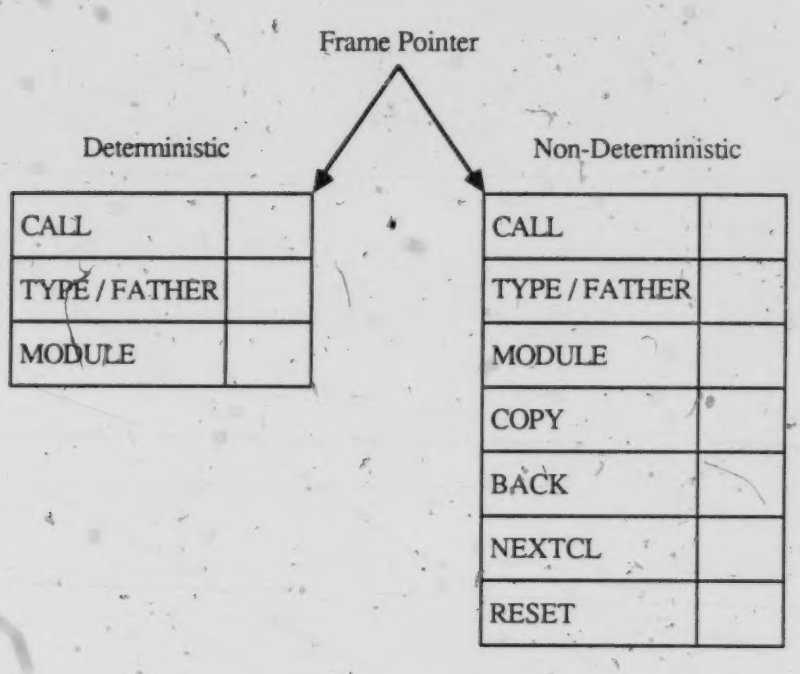


Figure 5.4: Deterministic and non-deterministic stack frames

5.2 Implementation of FLP

Incorporating extended unification requires modest modification to both the underlying data structures and the proof procedure of WUP. As a very first step, the PC_WORD representation of each rule is modified to include the target rewritten term (R.H.S.). A new data type 'RULE' is introduced

HEAD \ CALL	Free Var	Void Var	Int Const	Atom Const	Char Const	End List	Const List	Var List	Const Func	Var Func
Free Variable	FV	S	AH	AH	AH	AH	AH	CH	AH	CH
Void Variable	S	S	S	S	S	S	S	S	S	S
Integer Constant	AC	S	SC	F	F	F	F	F	F	F
Atom Constant	AC	S	F	SC	F	F	F	F	F	F
Character Constant	AC	S	F	F	SC	F	F	F	F	F
End List Constant	AC	S	F	F	F	SC	F	F	F	F
Constant List	AC	S	F	F	F	F	UL	UL	F	F
Variable List	AC	S	F	F	F	F	UL	UL	F	F
Constant Functor	CC	S	F	F	F	F	F	F	UF	UF
Variable Functor	AC	S	F	F	F	F	F	F	UF	UF

F - always Fail
S - always Succeed
FV - Free Variables
AH - Assign to Head
AC - Assign to Call
UL - Unify Lists
UF - Unify Functors
CH - Copy to Head
CC - Copy to Call
SC - Simple Comparison

Figure 5.5: Unification table

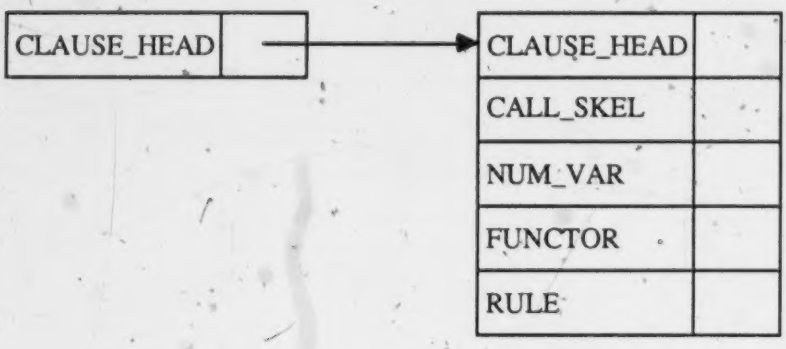


Figure 5.6: PC_WORD representation of a modified clause head

to differentiate it from normal subgoal call. As illustrated in Fig 5.6, the last PC_WORD points to the internal representation of a rewritten term. Similar to WUP, a stack node is pushed onto the runtime stack when the left hand side of a rewrite rule matches current call. After all subgoals of the condition are solved, rewriting proceeds to select as current call the right hand side rewritten term designated in the stack node. In the case of functor conflict, extended unification does not continue with the condition of the candidate clause. The usual inference procedure is suspended and a 'pseudo' stack node containing the suspended rule is pushed on top of the runtime stack. The pseudo node, also referred as conflicting node, denotes

a potential unification rather than matched clause-head pair. The next step is to locate the exact position of functor conflict. For example, if current call is f(d(X)) and the left hand side of a rule is f(e(X)), then they disagree at occurrence u = 1. Taking advantage from non-overlapping property, only the conflicting subterm of current call (i.e. d(X)) is narrowed. The conflicting subterm is now selected as the upcoming current call. Current call of this nature is labeled as a RULE rather than a normal subgoal call. After it is fully narrowed to its normal form, syntactic unification with the conflicting left hand side subterm (i.e. e(X)) is attempted. Since syntactic unification is carried out at a later stage, reference to the conflicting left hand side subterm must also be stored in the stack frame. Extended unification resumes with the suspended rule if syntactic unification succeeds, which signifies that the rewrite rule is really a candidate clause. Analogue to backtracking in trivial Prolog implementations, a global variable named 'CurConflict' represents the most recent conflicting node. A stack frame must accommodate sufficient information to retrieve the chain of successive conflicting nodes. Fig 5.7 shows the structure of an extended stack frame.

Minor adjustment of the proof procedure is required on two different portions of ABC algorithm: unification routine and handling of rewritten terms. The extended version distinguishes between syntactic unification and extended unification. The discrepancy is easily redressed by introducing a new boolean variable 'syntactic_unify'. If the variable is set to **true** when

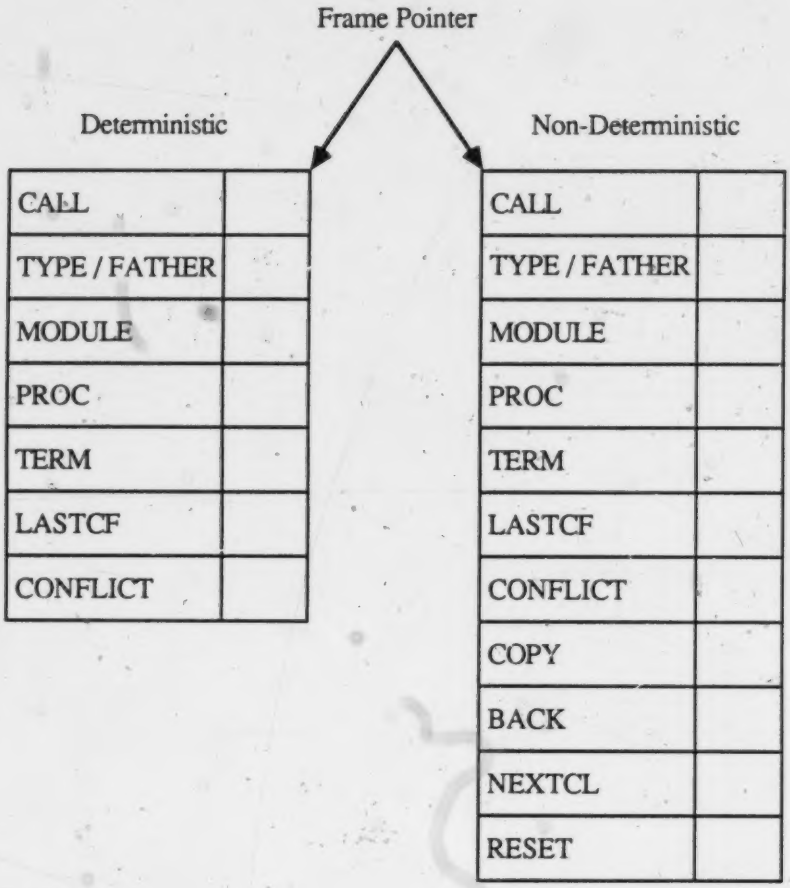


Figure 5.7: Modified version of runtime stack frames

functor conflict occurs, which stands for syntactic unification, the algorithm then returns with failure; otherwise, the inference procedure is suspended and a pseudo node is created. A simplified version of these changes is depicted in Fig 5.8.

where

CurCall	is the current call
tproc	is the suspended rewrite rule
ht	is the conflicting subterm of left hand side of tproc
ct	is the conflicting subterm of current call
RewriteRuleOf()	is a function returning right hand side of a rewrite rule
SetCurCall()	is a function setting the given argument as current call

As an example, assume the current call is f(d(X)) and the candidate clause is f(e(X)) : – g(X). Fig 5.9 shows the content of a simplified pseudo node after it is pushed onto the runtime stack.

Notice that the current call can now be a normal subgoal or a rewritten term of type RULE. Unification fails if a given subgoal has no candidate

```
if ( functor conflict )
  {
    if ( syntactic_unify )
      return(failure)

    /* Set up stack frame information */

    if ( more than one candidate clause )
      set up backtracking information (COPY,BACK,NEXTCL,RESET)

    set up compulsory information (FATHER,TYPE,MODULE)

    set up conflicting node information
      {
        CallOf(CurFrame) ← CurCall
        ProcOf(CurFrame) ← tproc
        RewriteTermOf(CurFrame) ← RewriteRuleOf(tproc)
        ConflictOf(CurFrame) ← ht
        LastCF(CurFrame) ← CurConflict
        CurConflict ← CurFrame
        SetCurCall(ct)
      }
  }
```

Figure 5.8: Simplified version of functor conflict handling

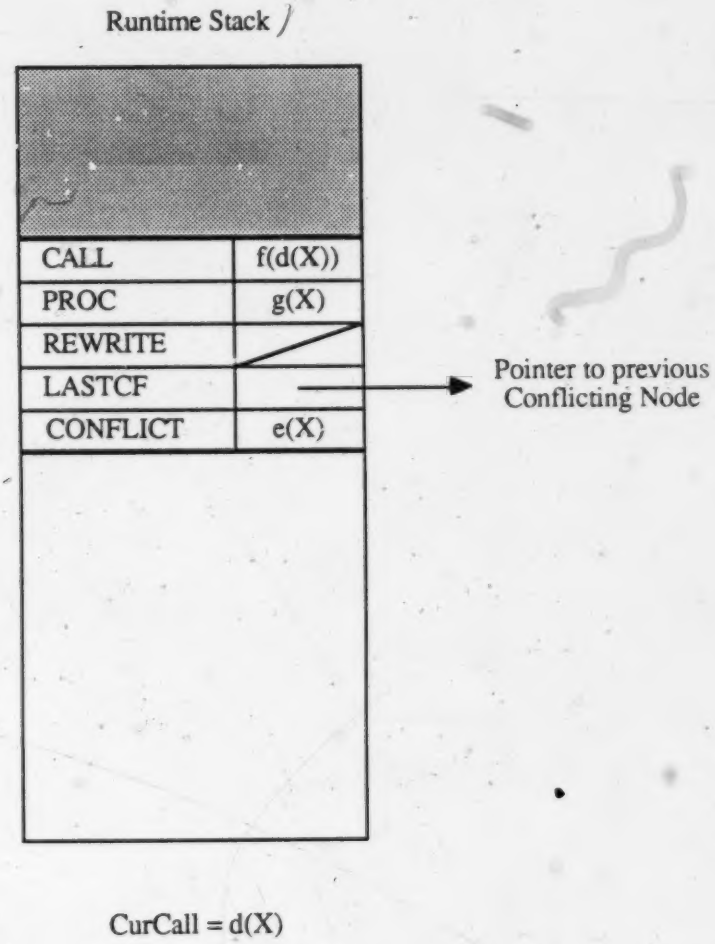


Figure 5.9: Pushing a pseudo stack node onto runtime stack

```
if ( unify )
  {
    if ( (IsRule(CurCall) && syntactic_unify )
      {
        CurProc ← ProcOf(CurFrame)
        syntactic_unify ← false
      }
  }
else if ( (IsRule(CurCall) && CurConflict )
  {
    syntactic_unify ← true
    unify(CurCall, ConflictOf(CurConflict))
  }
```

Figure 5.10: Modified proof procedure to handle rewritten terms

clause. A normal form is reached when no candidate clause responds to a rewritten term (of type RULE). Syntactic unification with the conflicting subterm, as indicated in the most recent conflicting node, is attempted thereafter. If syntactic unification succeeds, the most recent conflicting node represents a real matched clause-head pair. The process then continues with the suspended rewrite rule. An essential fraction of this modification is shown in Fig 5.10.

where

IsRule() is a function testing if the given argument is a rewritten term

ProcOf() is a macro for accessing the rewrite rule stored in the given frame

ConflictOf() is a macro for accessing the conflicting subterm stored in the given frame

A conflicting node is 'conceptually transformed' to a real stack node whenever its associated syntactic unification succeeds, or it is poped accordingly otherwise. Within this framework, modification of backtracking is next to none. A major consideration is retrieving the pointer to previous conflicting node when the most recent conflicting node is poped. Fig 5.11 is an example simulating the runtime stack for the following program.

Eg. 5.1.,

```
a(X) → b(X):- c(X)
f(b(X)):- d(X)
c(1)
d(X)
? f(a(X))
```

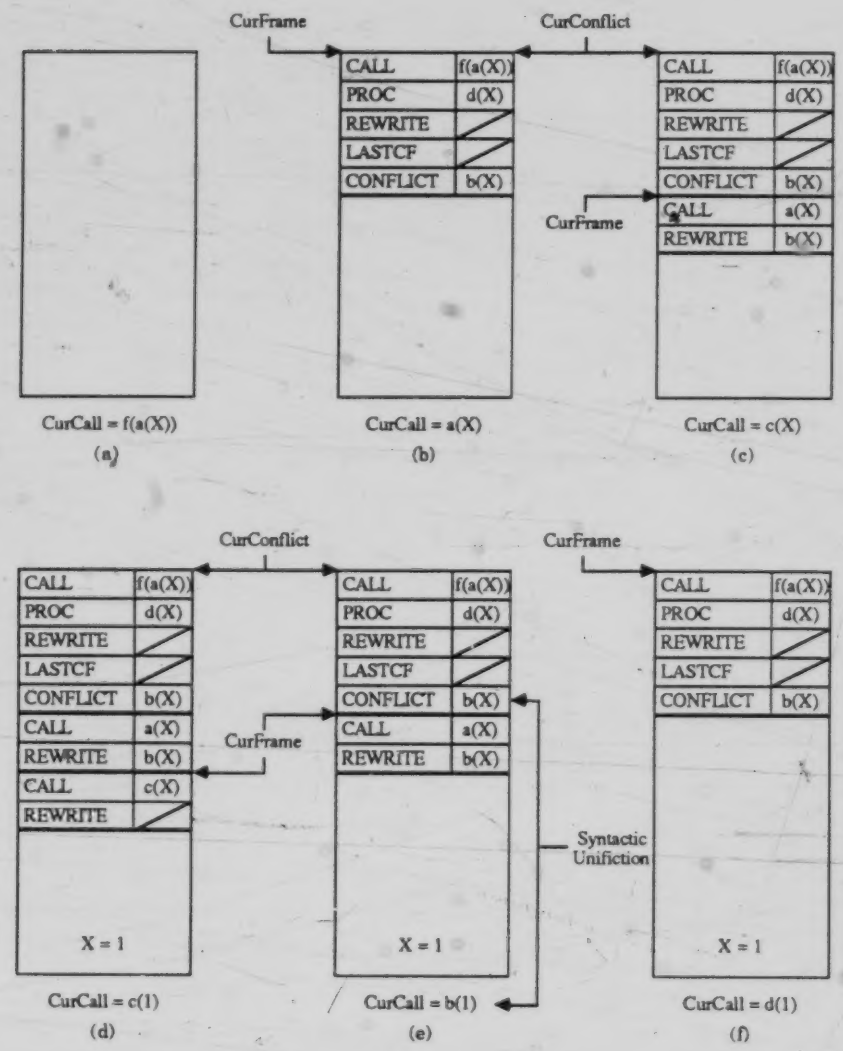


Figure 5.11: Runtime stack simulation of a program execution

Chapter 6

Conclusion

6.1 Summary

As a dialect of declarative languages, logic programming posses the distinct property of separating declarative semantics (what the output will be) from operational semantics (how to obtain the output). Despite its prominent features such as logic variables, bi-directional data flow and functional invertibility, the runtime behavior of non-deterministic logic programs is far too complicated and too difficult to control. Another major shortcoming of logic programming paradigm is the solo utilization of 'flat-natured' first-order constructor terms. Every term is syntactically equaled to itself only, but nothing else. Without function evaluation, higher-order programming is inhibited and the expressive power of logic languages is severely restricted.

In this thesis, we propose a conditional term rewriting system which serves as the bridge to combine logic programming and functional programming paradigms in a purely logical framework. The completeness and minimality characteristics of such system under a restricted subset of equality, namely E_c-equality, is further investigated. An important contribution of the proposed system is that an effective and efficient implementation scheme always exists and the scheme is readily portable to existing Prolog systems. We start by reviewing the basic concepts of logic programming and functional programming. Chapter 2 describes the similarities and differences among these two languages. Their discrepancies trace back to the use of different operational semantics on top of comparable logical frameworks. The key to successful amalgamation is addressed to two aspects: an equality theory rich enough to capture the underlying semantics of both paradigms and a suitable operational semantics for the united framework. Fundamental notions of pattern-matching reduction and unification-based resolution are presented in the first two chapters. Different strategies for a hybrid inference procedure, namely narrowing, are also introduced. Chapter 3 introduces E_c-equality which is a subset of classical equality theory. Restricted to a special class of confluent rewriting system, complete set of minimal E_c-unifiers always exists for two E_c-unifiable terms. Accompany with this framework is an effective and efficient narrowing strategy called outer-narrowing which enumerates the complete set of minimal E_c-unifiers. It is essentially a competent method

to interleave outermost narrowing with innermost narrowing strategies. A conditional term rewriting system is proposed in Chapter 4. This system, which is an extension of unconditional constructor-based rewriting system, is heavily inspired by the work of [You88]. We informally investigate the completeness characteristics of such system and claim that complete set of minimal E_c-unifiers always exists. We further suggest an extended unification scheme, mimicing the outer-narrowing strategy, as part of a complete inference procedure. The major advantages of this system are:

1. Complete set of minimal E_c-unifiers exists for two E_c-unifiable terms.

   Conditional narrowing is, in general, an incomplete inference procedure for unrestricted term rewriting systems. A serious impediment is the existence of non-normalizable solutions. In quest of completeness results, current research focuses on canonical (terminating and confluent) systems which alleviate the problem through the notion of unique canonical normal form. Although the confluence property is essential for correlating '$\stackrel{*}{\rightarrow}$' with '$=_E$', the terminating requirement seems to be too restrictive. Another area of research concentrates on the distinction between defined function symbols and constructors. A subset of classic equality is established between 'meaningful' or

'normalizable' terms. We informally show that conditional narrowing is complete for the proposed system, when confined to E_c-equality.

2. Efficient implementation of the proposed system is feasible.

   By the definition of E_c-equality, two terms are equaled iff they are (conditionally) narrowed to two unifiable constructor terms. Cross-unification between intermediate results is irrelevant. In terms of implementation, it is unnecessary to 'store' and 'cross-checking' all these intermediate terms. Substantial amount of memory are saved and the programming complexity is greatly reduced. Computation is relatively less expensive (in terms of space and time) than other similar systems.

Chapter 5 concentrates on the implementation issues of an extended unification scheme. Equally important is the fact that incorporating this extended scheme to existing Prolog systems requires minimal effort.

6.2 Future research

The conditional rewriting system proposed in this thesis can be used as a stepping stone to develop more general yet sophisticated systems. Future

research is addressed on two different but dependent areas:

1. Relax the conditions of the constructor-based system described in §4.1 to form another special class of rewriting system. An important aspect is the completeness and minimality of such system under E_c-equality. As a very first step, each rewrite rule can be generalized to

$$l \rightarrow r :- P \stackrel{*}{\rightarrow} c$$

with 'c' being a constructor term. A collection of these rules constitutes another instance of type III_n system. This system is less restrictive than our system in the sense that the condition is now narrowed to any constructor term. Future consideration can be directed towards confluent systems that are more general than type III_n system (for example, 'c' may not be a normal form).

2. Under E_c-equality, a term equals to itself only if it is narrowed to a constructor term. Irreducible function terms and the terms that are trapped in infinite derivations are considered 'undefined' in this framework. Another open area of research is to study the effect of a more general equality theory on these terms. [You89] introduces the notion of E_n-equality to manipulate irreducible function terms. The remaining question is how to handle terms that are trapped in non-terminating narrowing derivations. The suggested theory must

provide a coherent environment for all the cases. Ad hoc treatment of a particular case (such as irreducible term) is inadequate. The requirement of constructor-based conditional rewriting system may be banished if an elegant equality is discovered.

Alongside with these two areas of research is the search of a suitable inference mechanism. The operational semantics must be complete in the sense that complete set of minimal unifiers (if exist) is effectively enumerated. Further investigation is emphasized on a feasible and efficient implementation of such inference system.

Bibliography

[BG86] P.G. Bosco and E. Giovannetti. Ideal: An Ideal Deductive Applicative Language. In *Proceedings of IEEE 1986 Symposium on Logic Programming*, pages 89–95, 1986.

[BK86] J.A. Bergstra and J.W. Klop. Conditional Rewrite Rules: Confluence and Termination. *Journal of Computer and System Sciences*, 32:323–362, 1986.

[Cam84] J.A. Campbell, editor. *Implementations of Prolog*. Ellis Horwood, 1984.

[Che84] M.H.M Cheng. Design and Implementation of the Waterloo Unix Prolog Environment. Master's thesis, University of Waterloo, 1984.

[CL73] C.L. Chang and C.T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[CM79] R. Cartwright and J. McCarthy. *Recursive Programs as Functions in a First-Order Theory*, pages 576–607. Springer Verlag, 1979.

[DP88] N. Dershowitz and D.A. Plaisted. Equational Programming. *Machine Intelligence*, 11:21–56, 1988.

[DV87] M. Dincbas and P. Van Hentenryck. Extending Unification Algorithms for the Integration of Functional Programming into Logic Programming. *Journal of Logic Programming*, 4:199–227, 1987.

[Fay79] M. Fay. First Order Unification in an Equational Theory. In *Proceedings of the 4th Workshop on Automated Deduction*, pages 161–167, 1979.

[Fri85] L. Fribourg. SLOG: A Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting. In *Proceedings of IEEE 1985 Symposium on Logic Programming*, pages 172–184, 1985.

[GHT84] H. Glaser, C. Hankin, and D. Till. *Principles of Functional Programming.* Prentince Hall, 1984.

[GM84] J.A. Goguen and J. Meseguer. Equality, Types, Modules and (why not?) Generics for Logic Programming. *Journal of Logic Programming*, 2:179–210, 1984.

[GM86] E. Giovannetti and C. Moiso. A Completeness Result for E-Unification Algorithms Based on Conditional Narrowing. In *Proceedings of Foundations of Logic and Functional Programming Workshop*, pages 157–167, 1986.

[GM87] E. Giovannetti and C. Moiso. Notes on the Elimination of Conditions. In *Proceedings of 1987 International Workshop on Conditional Term Rewriting Systems*, pages 91–97, 1987.

[HO80] G. Huet and D.C. Oppen. *Equations and Rewrite Rules: A Survey*, pages 349–405. Academic Press, 1980.

[HO82] C.M. Hoffmann and M.J. O'Donnell. Programming with Equations. *ACM Transactions on Programming Languages and Systems*, 4:83–112, 1982.

[Hog84] C.J. Hogger. *Introduction to Logic Programming.* Academic Press, 1984.

[Hul80] J.M. Hullot. Canonical Forms and Unification. In *Proceedings of the 5th International Conference on Automated Deduction*, pages 318–334, 1980.

[JL87] J.Jaffer and J.L. Lassez. Constraint Logic Programming. In *Conference Record of the 14th ACM Symposium on Principles of Programming Langauages*, pages 111–119, 1987.

[Kap84] S. Kaplan. Fair Conditional Term Rewriting Systems: Unification, Termination and Confluence. Technical Report 194, University of Orsay, 1984.

[Kor83] W.A. Kornfeld. Equality for Prolog. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, pages 514–519, 1983.

[Kow74] R.A. Kowalski. Predicate Logic as a Programming Language. In *1974 IFIP Congress Proceedings*, pages 569–574, 1974.

[Kow79] R.A. Kowalski. Algorithm = Logic + Control. *Communication of ACM*, 22(7):424–436, 1979.

[Llo84] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[LPB+87] G. Levi, C. Palamidessi, P.G. Bosco, E. Giovannetti, and C. Moiso. A Complete Semantic Characterization of K-LEAF, a Logic Language with Partial Functions. In *Proceedings of IEEE 1987 Symposium on Logic Programming*, pages 318–327, 1987.

[O'D85] M.J. O'Donnel. *Equational Logic as a Programming Language.* MIT Press, 1985.

[Plo72] G.D. Plotkin. Build-in Equatoional Theories. *Machine Intelligence*, 7:73–90, 1972.

[Ret87] P. Rety. Improving Basic Narrowing Techniques. In *Proceedings of 1987 International Conference on Rewriting Techniques and Applications*, pages 228–241, 1987.

[Rob65] J.A. Robinson. A Machine-oriented Logic Based on the Resolution Principle. *Journal of ACM*, 12(1):23–41, 1965.

[vE82] M.H. van Emden. An Algorithm for Interpreting Prolog Programs. In *Proceedings of the 1st International Conference on Logic Programming*, 1982.

[vEK76] M.H. van Emden and R.A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of ACM*, 23(4):733–742, 1976.

[vEL84] M.H. van Emden and J.W. Llyod. A Logical Reconstruction of Prolog II. *Journal of Logic Programming*, 2:143–149, 1984.

[vEY87] M.H. van Emden and K. Yukawa. Logic Programming with Equations. *Journal of Logic Programming*, 4:265–288, 1987.

[Wir82] N. Wirth. *Programming in Modula-2.* Springer Verlag, 1982.

[You88] J-H. You. Outer Narrowing for Equational Theories Based on Constructors. In *Procedings of 15th International Colloquium on Automata, Languages and Programming*, pages 727–741, 1988.

[You89] J-H. You. Unification Modulo an Equational Theory for Equational Logic Programming. Technical Report TR 89-5, University of Alberta, 1989.